Universidade Estadual de Campinas
Faculdade de Engenharia Mecânica
Engenharia de Controle e Automação

ES952 - TRABALHO DE GRADUAÇÃO II

“ANÁLISE DO USO DE UM SISTEMA OPERACIONAL DE TEMPO REAL EM UM SOFTWARE DESENVOLVIDO PARA UM CONTADOR CRESCENTE/DECRESCENTE COM RESOLUÇÃO DE 100 ms.”

Aluno: Gabriel Soares Martins **RA:** 023876
Orientador: Prof. Dr. Luiz Otávio Saraiva Ferreira

JUNHO - 2009

GABRIEL SOARES MARTINS

"ANÁLISE DO USO DE UM SISTEMA OPERACIONAL DE TEMPO REAL EM UM SOFTWARE DESENVOLVIDO PARA UM CONTADOR CRESCENTE/DECRESCENTE COM RESOLUÇÃO DE 100 ms."

Trabalho de conclusão apresentado para a banca examinadora do curso de Engenharia de Controle e Automação da Universidade Estadual de Campinas, como exigência parcial para a obtenção do título de Engenheiro de Controle e Automação, sob orientação do Prof. Dr. Luiz Otavio Saraiva Ferreira.

Campinas, 2009

RESUMO

Este trabalho de graduação fez uma comparação entre dois softwares utilizados para uma mesma aplicação: um contador crescente/decrescente. Em um dos softwares foi utilizado um sistema operacional de tempo real. Através desta comparação, foi mostrado que o uso sistema operacional de tempo real aumentou em quase 10 vezes a precisão da contagem e reduziu o número de funções e variáveis do código necessárias para esta aplicação.

Palavras-chave: sistemas operacionais de tempo real, tarefas, semáforos, filas de mensagens, escalonador, sistemas de tempo real, kernel, PIC, µC/OS-II, sistemas embarcados.

ABSTRACT

This work compared two different softwares for the same application: an up/down counter. One of them used a real time operating system. With the comparison, it was shown that the real time operating system utilization increased almost 10 times the counting precision and reduced the number of code´s functions and variables needed by the application.

Key-words: real time operating system, tasks, semaphores, message queues, scheduling, real time systems, kernel, PIC, µC/OS-II, embedded systems.

Sumário

1.INTRODUÇÃO

A sociedade atual tem apresentado uma quantidade crescente de aplicações que apresentam comportamentos definidos segundo restrições temporais. O momento exato em que uma atividade deve ser executada, bem como o momento em que ela deve parar sua execução se tornou muito importante para o desenvolvimento tecnológico.

Aplicações com restrições temporais se encontram no controle de plantas industriais, de tráfego aéreo ou ferroviário, de aquisição de dados, nas telecomunicações, na eletrônica embarcada em carros e aviões, na robótica, equipamentos médicos, em sistemas de multimídia, etc. Estas aplicações, sujeitas a restrições temporais, são comumente identificadas como Sistemas de Tempo Real.

Os sistemas operacionais de tempo real (RTOS) surgem neste cenário como uma importante ferramenta para trazer confiabilidade a estes sistemas, bem como para facilitar o desenvolvimento de aplicações nas áreas citadas.

Este trabalho tem como objetivo fazer uma análise comparativa entre dois softwares (com, e sem, o uso de um RTOS) para evidenciar as melhorias que estes tipos de sistemas operacionais fornecem no desenvolvimento de uma aplicação.

2.DEFINIÇÃO DE SISTEMAS EMBARCADOS

Em (LACERDA, 2006), um sistema embarcado é definido como uma combinação de componentes de hardware e software, que usa interfaces de entrada e saída específicas e dedicadas para realizar uma determinada atividade. Um sistema embarcado é considerado um sistema computacional de uso específico, que interage continuamente com o ambiente a sua volta, por meio de sensores e atuadores, utilizando seus recursos computacionais, projetados restritamente para executar a função para qual foi desenvolvido.

Ao contrário de programas desenvolvidos para computadores de uso geral, onde um único programa pode realizar tarefas diferentes, o software embarcado geralmente não pode ser usado em outro sistema embarcado sem que antes sejam feitas mudanças significativas. Isso porque cada hardware é projetado sob medida para uma dada aplicação, ou seja, o projeto do hardware e do software de um sistema embarcado é feito em conjunto, sendo, portanto, interdependentes.

As principais unidades funcionais de um hardware embarcado, segundo (BITTON, 2008), são os processadores, cuja principal função é processar instruções e dados. Um processador funciona como o dispositivo central de controle de um hardware embarcado, sendo responsável por gerenciar dispositivos de memória, barramentos e I/Os (entradas e saídas).

A complexidade de um processador geralmente determina se este é classificado como um microprocessador (MPU) ou um microcontrolador (MCU). Um microcontrolador contém as mesmas funcionalidades básicas de um computador, tudo incluso em um único chip de silício. Conforme ilustra a **Figura 1**, um

microcontrolador é dividido em 4 blocos principais: processador (CPU); memória; I/O (entradas/saídas); e periféricos.

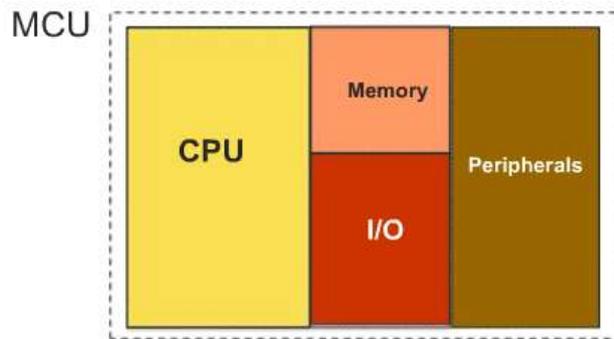



Figura 1. Ilustração em blocos de um microcontrolador. (BITTON, 2008).

O processador executa funções matemáticas e lógicas, nas quais estão os operandos e as operações necessárias para a execução do programa. O processador também lê e armazena os dados. A memória é dividida em: ROM (*Read-only-memory*), que é a memória onde o microcontrolador armazena o código do programa; e RAM (*Random-access-memory*), que é a memória onde o MCU armazena os dados temporários e outros dados que se alteram durante a execução do programa, como variáveis e dados intermediários. O terceiro bloco do MCU são os seus pinos de entrada e saída, através dos quais o microcontrolador envia ou recebe dados do seu ambiente externo. O quarto bloco são os periféricos que o MCU possui em seu chip. Estes periféricos podem ser temporizadores/contadores, PWMs, conversores analógico-digitais, conversores digitais-analógicos, dentre outros, que serão as ferramentas que permitirão ao desenvolvedor alcançar os requisitos de sua aplicação. Existem diversos tipos de periféricos que diferenciam um MCU de outro.

A diferença de um microcontrolador e um microprocessador está no fato de que em um microprocessador a memória principal e os periféricos são externos ao dispositivo. Além disso, o microprocessador necessita de barramentos de dados e de endereçamento para se comunicar com os recursos externos a seu processador. A Figura 2 ilustra um microprocessador.

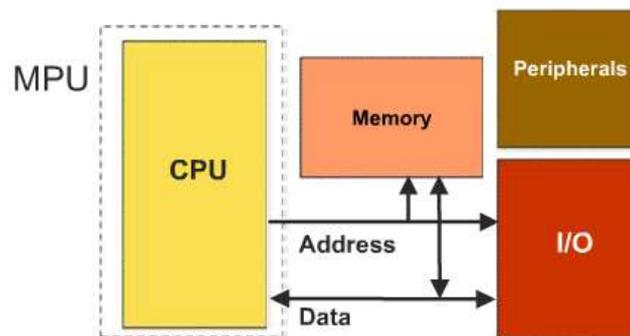



Figura 2. Ilustração em blocos de um microprocessador. (BITTON, 2008).

Pelo fato de um sistema embarcado realizar um conjunto de tarefas pré-definidas, geralmente com requisitos específicos, o sistema pode ser otimizado, por exemplo, reduzindo seu tamanho, recursos computacionais e custo do produto.

Diversos produtos utilizados na vida rotineira de uma pessoa são sistemas embarcados. A Figura 3 mostra alguns exemplos destes sistemas:

Figura 3. Exemplos de sistemas embarcados. (LACERDA, 2006).

3.SISTEMAS DE TEMPO REAL

Sistemas de tempo real podem ser definidos, de acordo com (LI e YAO, 2003), como sistemas que respondem a eventos externos, síncronos ou assíncronos, atendendo a restrições de tempo especificadas. Responder a um evento externo inclui: reconhecer quando o evento acontece; executar o processamento necessário para o resultado do evento; e fornecer como saída o resultado deste processamento. Todas estas ações devem ser executadas dentro de um intervalo de tempo especificado. A Figura 4 mostra, de maneira simplificada, um sistema de tempo real.

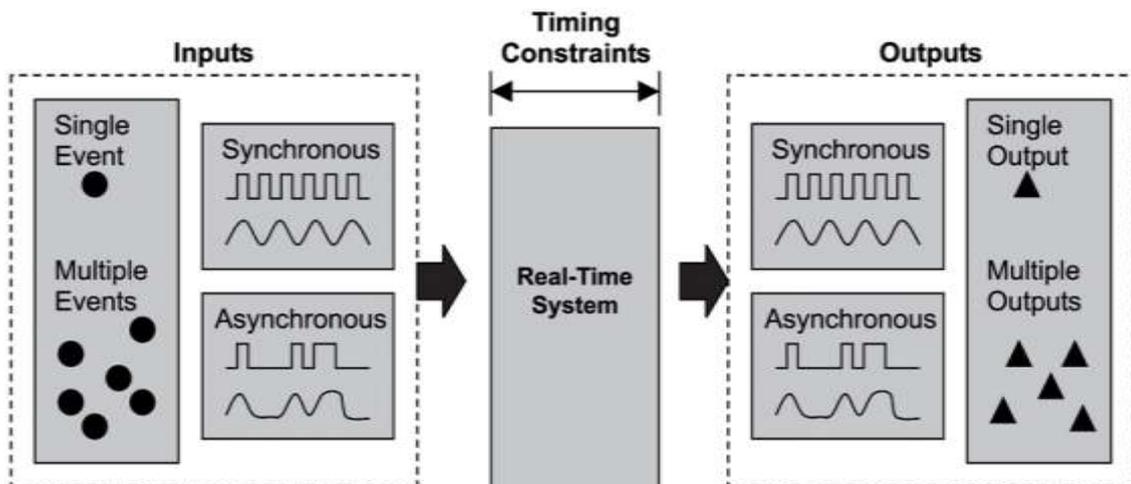



Figura 4. Uma visão simples de um sistema de tempo real. (LI e YAO, 2003).

Um sistema de tempo real deve produzir resultados computacionais corretos, ou seja, ter como característica, exatidão funcional ou lógica. Além disso, estes resultados devem ser computados dentro de um tempo pré-definido, portanto, o sistema deve ter exatidão temporal. É importante ressaltar que a exatidão completa de um sistema de tempo real depende tanto da exatidão funcional quanto da exatidão temporal do mesmo.

3.1. Tipos de sistemas de tempo real

Conforme (LI e YAO, 2003), em alguns sistemas de tempo real, a exatidão funcional pode ser sacrificada em função da busca pela exatidão temporal. Esta possibilidade nos permite dividir os sistemas de tempo real em duas classes: sistemas de tempo real do tipo crítico (*hard*) e do tipo brando (*soft*).

Sistemas de tempo real crítico devem ser executados nos limites de tempos especificados, com grau de flexibilidade próximo a zero. De acordo (GANSSLE, 1998), estes limites de tempo podem ser intervalos periódicos, um tempo em particular ou a chegada de um evento. Estes sistemas falham, por definição, se estes limites de tempo não são atendidos. Segundo (STEWART, 2001), estas falhas incluem danos a equipamentos, grandes perdas de investimento, e até ferimentos ou mortes dos usuários do sistema. Um exemplo de sistema de tempo real crítico é um controlador de vôo. Se uma ação, em resposta a um novo evento, não for executada dentro do tempo especificado, pode haver uma desestabilização do vôo e levar a um acidente.

Em contrapartida, de acordo com (GANSSLE, 1998) sistemas de tempo real do tipo brando (*soft*) são definidos como sistemas que não são do tipo crítico, mesmo ainda possuindo restrições de tempo para sua execução. Para estes tipos de sistema, não atender, desde que esporadicamente, seus limites de tempo não compromete a integridade do sistema, apesar de que o custo do sistema pode aumentar consideravelmente de acordo com seu *delay*. Em (STEWART, 2001) tem-se como exemplo de um sistema de tempo real brando o piloto automático de um automóvel. Suponha que o software não conseguiu medir a tempo a velocidade

atual do automóvel para o algoritmo de controle utilizá-la na manutenção da velocidade especificada pelo usuário. Mesmo com a informação perdida, o sistema não será afetado, pois o valor da velocidade adquirida em amostras consecutivas não é discrepante suficientemente para danificar o sistema. No entanto se o sistema perder várias amostras consecutivas da velocidade do carro, ele estará comprometido, pois o piloto automático não conseguirá manter a velocidade especificada dentro de um erro aceitável. Portanto, de acordo com (LI e YAO, 2003), sistemas de tempo real do tipo brando devem atender seus limites de tempo, mas com um grau de flexibilidade maior do que os do tipo crítico.

3.2. Principais características dos sistemas de tempo real

Segundo (GANSSLE et al, 2008), as principais características que os sistemas de tempo real devem apresentar são:

- **Alto nível de escalonamento** – os requisitos de tempo do sistema devem ser satisfeitos mesmo com um alto uso dos recursos do mesmo. Isto implica que o sistema deve ter a habilidade de todas suas tarefas atenderem aos seus limites de tempo especificados.
- **Operabilidade em péssima latência –** o sistema deve operar, seguramente, mesmo nos piores tempos de resposta aos eventos.
- **Estabilidade sob sobrecarga transiente** – quando o sistema está sobrecarregado por eventos e for impossível atender a todos os limites de tempo de suas tarefas, os limites mais críticos devem ser garantidos.

Outra importante característica de um sistema de tempo real é a previsibilidade do mesmo. Segundo (STEWART, 2001), o termo previsibilidade refere-se aos sistemas que possuem um comportamento no tempo, dentro de uma faixa aceitável, que atendem a um padrão. Geralmente, o pior limite de tempo de execução de cada tarefa deve ser conhecido para se criar um sistema previsível.

Um tipo especial de sistema previsível é um sistema determinístico. Nestes tipos de sistemas, não somente o comportamento no tempo deve atender a um padrão, como ele pode ser pré-determinado, ou seja, uma condição inicial pode ser dada ao sistema, determinando seu estado futuro. Um sistema de tempo real torna-se determinístico quando uma fatia de tempo é pré-alocada para a execução de cada tarefa.

4.UMA BREVE DEFINIÇÃO DE SISTEMA OPERACIONAL

Conforme (LI e YAO, 2003), aplicações simples de software são geralmente designadas para serem executadas seqüencialmente, uma instrução de cada vez, numa cadeia pré-determinada de instruções. Entretanto, este esquema é inapropriado para aplicações de sistemas embarcados de tempo-real, que geralmente tratam de várias entradas e saídas dentro de curtos intervalos de tempo.

Softwares para aplicações em sistemas embarcados de tempo real devem ser desenvolvidos para serem concorrentes. Este desenvolvimento concorrente requer que os desenvolvedores decomponham a aplicação em unidades de programas pequenas, escalonáveis e seqüenciais. Dessa forma o sistema pode trabalhar num ambiente multitarefa e alcançar o desempenho e tempo necessário para atender aos requisitos do sistema de tempo real.

Quanto mais complexo for o sistema, mais tarefas ele terá e mais complexas elas serão. Num sistema multitarefa, várias tarefas requisitarão tempo de CPU para sua execução e como há apenas uma CPU[1], alguma organização e coordenação serão necessárias para que cada tarefa tenha o tempo de CPU suficiente para sua execução. É no gerenciamento desta organização e coordenação de tarefas que se enquadram os sistemas operacionais.

Em (GANSSLE et al, 2008), sistemas operacionais (SOs) são definidos como um conjunto de bibliotecas de software que atendem dois propósitos: prover um nível de abstração para o software que o torna menos dependente do hardware e

[1] Atualmente existem processadores que possuem mais CPUs, como é o caso dos processadores multi-cores que chegam a ter até quatro núcleos (CPUs). No entanto o número de tarefas requisitando uma CPU para sua execução ainda é muito superior do que o número de núcleos, fazendo com que elas ainda concorram pelos núcleos existentes.

gerenciar os vários recursos de hardware e software para assegurar uma total eficiência e confiabilidade da aplicação.

Todo SO possui um kernel. O kernel é o componente que contém as principais funcionalidades do SO, que incluem:

- **Gerenciamento de processos** – como o SO gerencia e visualiza outro software no sistema;
- **Gerenciamento de memória** – o espaço de memória do sistema é compartilhado por todos os diferentes processos, necessitando de um gerenciamento do acesso e da alocação do espaço de memória;
- **Gerenciamento dos sistemas de entrada/saída** – como os dispositivos de entrada e saída são compartilhados por diferentes processos, é necessário um gerenciamento destes dispositivos para que os processos os utilizem de maneira correta.

O gerenciamento de processos é o subsistema central de um SO. Todos os outros subsistemas são dependentes deste. A forma como o kernel gerencia seus processos ou tarefas para que eles utilizem a CPU é denominada escalonamento (ou agendamento) de tarefas ou processos. O escalonador (ou agendador) é o processo que trata do escalonamento e está intimamente relacionado com os seguintes fatores:

- Utilização da CPU (deve-se buscar a máxima utilização da CPU);
- Número de processos executados por unidade de tempo;
- Quantidade de tempo de espera de uma tarefa pronta pra execução;
- Tempo de resposta de um evento;
- Quantidade de tempo que cada tarefa terá para usar a CPU.

4.1.Grupos de sistemas operacionais – GPOSs e RTOSs

Em (LI e YAO, 2003) são apresentados dois grupos de sistemas operacionais: GPOSs (General Purpose Operanting Systems), ou sistemas operacionais de propósito geral, e RTOS (Real Time Operating Systems), ou sistemas operacionais de tempo real. Estes dois grupos possuem as seguintes similaridades e diferenças:

- Similaridades entre RTOSs e GPOSs:
  - Algum nível de execução multitarefa;
  - Gerenciamento dos recursos de hardware e software;
  - Fornecimento de serviços básicos do OS para as aplicações do desenvolvedor;
  - Abstração do hardware através do software.
- Funcionalidades chave dos RTOS que os diferem dos GPOSs:
  - Maior confiabilidade em aplicações embarcadas;
  - Habilidade de inserção ou remoção de módulos para alcançar as necessidades do sistema;
  - Melhor desempenho;
  - Requisitos de memória reduzidos;
  - Políticas de escalonamento específicas para sistemas embarcados de tempo real;
  - Melhor adaptação em diferentes plataformas;
  - Possibilidade dos executáveis do OS inicializarem e rodarem o sistema a partir de suas memórias RAM e ROM.

Uma das principais diferenças entre um RTOS e um GPOS está na sua forma de escalonar as tarefas. De acordo com (LEROUX e SCHAFFER, 2006), um GPOS usa uma política mais justa para fornecer tempo de CPU às tarefas e processos, tentando fornecer a mesma fatia de tempo da CPU para os mesmos. Tal política aumenta o número de processos executados por unidade de tempo, mas não assegura que tarefas mais críticas (de alta prioridade) sejam executadas preferencialmente às de baixa prioridade. Um GPOS pode ainda diminuir a prioridade de uma tarefa para ajustá-la a política de igualdade, e fornecer tempo de CPU a outras tarefas do sistema. Dessa forma uma tarefa de alta prioridade pode sofrer preempção, ou seja, perder o controle da CPU, durante sua execução, para uma tarefa de menor prioridade.

Na maioria dos GPOSs, o kernel não é preemptivo. Conseqüentemente, uma tarefa de alta-prioridade do desenvolvedor da aplicação nunca fará a preempção de uma chamada do kernel[2], mas deverá esperar pela execução completa da chamada – mesmo se esta for o processo de menor prioridade em todo o sistema. GPOSs não possuem limites para aumentar a latência de sua troca de contexto: quanto mais tarefas existir no sistema, mais tempo levará para a CPU executar a mesma tarefa novamente. Todas estas características podem fazer com que tarefas de alta prioridade não alcancem seus limites de tempo.

Segundo (LEROUX e SCHAFFER, 2006), num RTOS, as tarefas são executadas por ordem de prioridade. Se uma tarefa de alta prioridade fica pronta para execução, ela irá, num pequeno e limitado intervalo de tempo, tomar o controle

[2] Uma chamada do kernel, também conhecida por chamada de sistema, é uma requisição feita pelo aplicativo ao sistema operacional, para que o kernel execute uma ação, pertencente a um conjunto pré-definido de tarefas, que a aplicação não possa executar por não ter a permissão adequada para executá-la durante seu próprio fluxo de execução. As chamadas de sistema fornecem uma interface entre os processos e o sistema operacional.

da CPU de qualquer outra tarefa de menor prioridade que esteja, inclusive, em execução. Mais ainda, ela pode ininterruptamente ser executada pela CPU até o seu fim ou até que sofra preempção de outra tarefa de maior prioridade. Esta política conhecida como escalonamento preemptivo baseado em prioridade (aqui chamado de escalonamento preemptivo-prioritário) permite seguramente que as tarefas de maior prioridade alcancem seus limites de tempo, não importando quantas tarefas estejam competindo pela CPU.

Conforme (LI e YAO, 2003), hoje, GPOSs rodam predominantemente em sistemas como, computadores pessoais, workstation e mainframes. Em alguns casos, GPOSs rodam em sistemas embarcados que possuem memória ampla e requisitos de tempo real que não trazem problemas caso não sejam atendidos. GPOSs, geralmente, requerem muita memória e, portanto, não são apropriados para sistemas embarcados que possuem memória limitada e requisitos de alto desempenho. RTOSs por outro lado atendem estes requisitos, sendo confiáveis, compactos e possuindo bom desempenho em sistemas embarcados de tempo real.

5.SISTEMAS OPERACIONAIS DE TEMPO REAL – RTOS

Um sistema operacional de tempo real (RTOS) é, de acordo com (LI e YAO, 2003), um programa que agenda as execuções de suas tarefas de forma temporal, gerencia os recursos de sistemas, e fornece uma base consistente para desenvolver códigos para aplicações de tempo real. Segundo (GANSSLE et al, 2008), um RTOS deve ter as seguintes características para ser considerado como tal:

- Ser multitarefa;
- Ser preemptivo;
- Suportar prioridade de tarefas;
- Suporta mecanismos de sincronização de tarefas;
- Limitar as condições de inversão de prioridade das tarefas;
- Ter seu comportamento conhecido para que o desenvolvedor do software consiga prever com precisão o desempenho do sistema.

Um RTOS pode ser composto apenas de um kernel, que é o núcleo supervisor do software, fornecendo algoritmos de mínima lógica, agendamento e gerenciamento de recursos. Pode também conter uma combinação de vários módulos incluindo kernel, sistemas de arquivos, protocolos de rede e componentes necessários para uma aplicação em particular. Uma ilustração em alto-nível de um RTOS é apresentada na Figura 5.

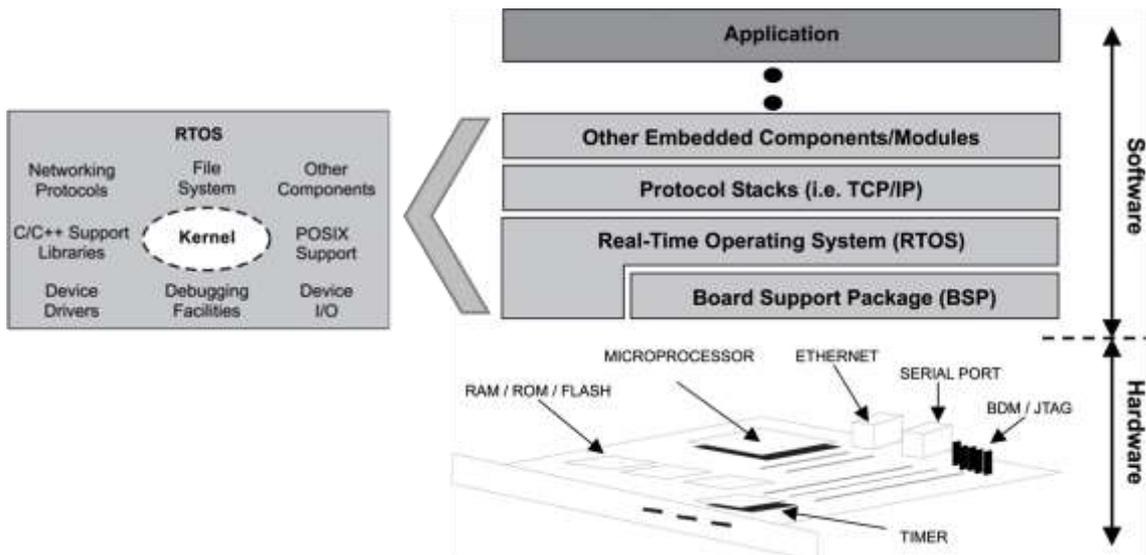



Figura 5. Visão em alto nível de um RTOS (LI e YAO, 2003).

O kernel de um RTOS geralmente contém os seguintes componentes (Figura 6):

- **Escalonador** – consiste de um conjunto de algoritmos que determina qual tarefa, e quando, será executada.
- **Objetos** – são construções especiais do kernel que facilitam a criação de aplicações. Dentre esses objetos estão incluídos as tarefas, os semáforos, as caixas de correio, as filas de mensagens e etc.
- **Serviços –** são as operações que o kernel executa sobre os objetos, os serviços de gerenciamento do tempo, de gerenciamento de recursos e etc.

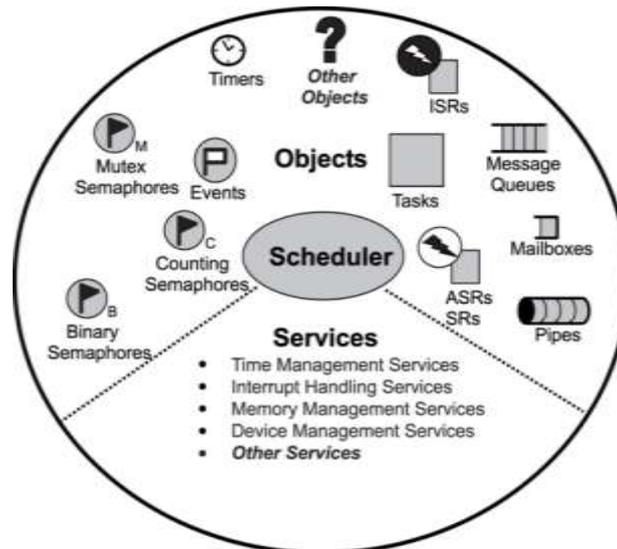



Figura 6. Componentes do kernel de um RTOS. (LI e YAO, 2003).

5.1.O Escalonador

O escalonador é, de acordo com (LI e YAO, 2003), a principal ferramenta do kernel. Ele é o responsável por fornecer os algoritmos que irão determinar qual tarefa, e quando, será executada.

5.1.1.Entidades escalonáveis

Conforme (LI e YAO, 2003), uma entidade escalonável é um objeto do kernel que compete pelo tempo de execução (tempo de utilização da CPU) em um sistema, baseado num algoritmo pré-definido de escalonamento. As tarefas são as entidades escalonáveis encontradas na maioria dos kernels.

Uma tarefa é uma função independente, cuja execução se baseia numa seqüência de instruções independentemente escalonáveis. É importante deixar claro que mensagens e semáforos, apesar de serem objetos, não são considerados

entidades escalonáveis. Estes itens são considerados objetos usados para comunicação e sincronização entre tarefas.

Um conjunto de tarefas a ser executadas define um ambiente multitarefa.

5.2.Tratamento das tarefas num ambiente multitarefa.

Num ambiente multitarefa, o kernel trata as tarefas de forma que todas aparentam estar sendo executadas simultaneamente. No entanto, o kernel está realizando o agendamento das tarefas (escolhendo qual será a próxima tarefa a ser executada) e trocando a CPU entre as várias tarefas existentes baseado no algoritmo de agendamento utilizado. O escalonador ou agendador deve assegurar que a tarefa certa seja executada na hora certa, ou seja, no limite de tempo determinado.

À medida que o número de tarefas a serem agendadas aumenta, os requisitos de desempenho do CPU também aumentam. Isto ocorre devido há um aumento no número de trocas de contexto das diferentes tarefas a serem executadas.

5.3.Interrupções e o tratamento de ISRs num ambiente multitarefa

Rotinas de interrupções (ISR), diferentemente de tarefas, não seguem os algoritmos de escalonamento, mas são executadas a partir de interrupções ocorridas em hardware e, geralmente, são prioritárias em relação às tarefas. Uma interrupção é, segundo (LABROSSE, 2002), um mecanismo de hardware usado para informar a CPU que um evento assíncrono ocorreu. Quando uma interrupção é reconhecida, a CPU pára seu fluxo normal de execução, carrega os endereços de retorno dos

conteúdos de seus registradores na pilha, e pula para um endereço específico (determinado pelo fabricante) de memória de programa, relacionada ao sinal de interrupção recebido, onde estará codificada a rotina de serviço da interrupção (ISR) ou rotina de interrupção. Quando esta rotina processa o evento e completa sua execução, o programa retorna para a tarefa agendada pelo escalonador para ser executada.

5.4.Troca de contexto

Cada tarefa tem seu próprio contexto, que, de acordo com (LI e YAO, 2003), é o estado em que estão os registradores da CPU quando uma determinada tarefa está em execução. Quando outra tarefa entra em execução, este estado é salvo na pilha da tarefa que saiu de execução e é requisitado pela CPU toda vez que esta tarefa é agendada novamente para ser executada. Segundo (LAMIE, 2009), o contexto de uma tarefa inclui seu conjunto de registradores, o contador de programa (PC) e informações críticas relacionadas às tarefas.

Toda vez que uma tarefa é criada, o kernel cria e mantém associado a ela uma estrutura chamada de *task control block* (TCB), ou bloco de controle da tarefa. Estes TCBs são estruturas de dados do sistema, os quais o kernel usa para guardar informações específicas de cada tarefa. Os TCBs contêm tudo que o kernel precisa saber sobre uma tarefa em particular. Quando uma tarefa está em execução, seu contexto, altamente dinâmico, é mantido no TCB. Quando uma tarefa não está em execução, seu contexto está estático dentro do TCB, pronto para ser restaurado na próxima vez que a tarefa entrar em execução.

Uma troca de contexto ocorre quando o escalonador troca uma tarefa por outra e consiste basicamente de salvar os registradores na pilha da tarefa a ser suspensa e restaurar os registradores da pilha da tarefa a ser executada. A Figura 7, ilustra uma troca de contexto.

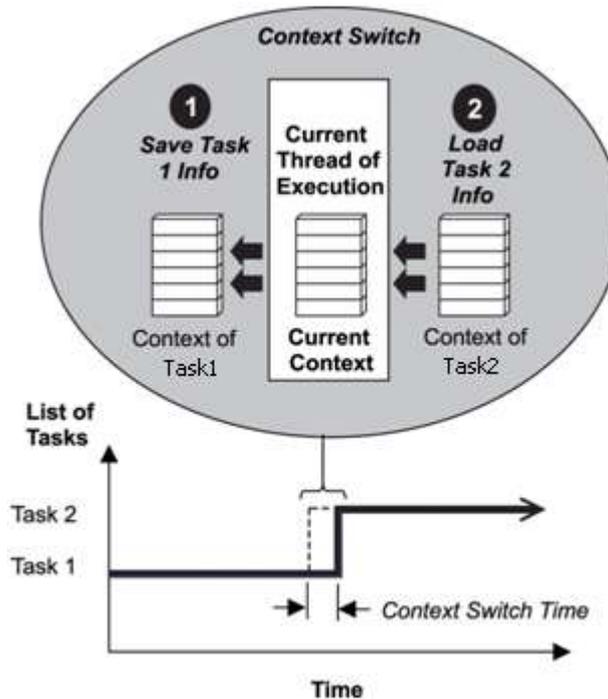



Figura 7. Troca de contexto entre tarefas. (LI e YAO, 2003).

Conforme mostrado na Figura 7, quando o escalonador do kernel determina que seja necessário parar a tarefa em execução (task 1), e começar a executar a task 2, o kernel, primeiramente, salva o contexto da task 1 no TCB desta tarefa. Depois, ele carrega o contexto da task 2 a partir do TCB desta tarefa, tornando-a a tarefa em execução. O contexto da task 1 é então “congelado” enquanto a task 2 é executada. Se o escalonador precisar executar a task 1 novamente, esta tarefa irá continuar sua execução a partir da posição em que sofreu a troca de contexto.

Conforme (LAMIE, 2009) os passos de uma troca de contexto estão representados na **Erro! Fonte de referência não encontrada.**. O número de ciclos de instrução que cada etapa leva para ser executada é dependente do RTOS utilizado e totaliza de 50 a 500 ciclos de instrução para se realizar uma troca de contexto. O tempo que o escalonador leva para trocar de uma tarefa para outra é o tempo de troca de contexto e é proporcional ao número de ciclos de instrução para realizá-la.

Tabela 1. Seqüência de passos de uma troca de contexto. (LAMIE, 2009).

Passo	Operação	Ciclos
1	Salvar o contexto (ie: os valores dos registradores FP e GP e o PC) da tarefa a ser suspensa na pilha;	20-100
2	Salvar o atual ponteiro da pilha no TCB da tarefa a ser suspensa;	2-20
3	Trocar para o ponteiro da pilha do sistema;	2-20
4	Retornar ao escalonador;	2-20
5	Encontrar a tarefa de maior prioridade que está pronta para ser executada;	2-50
6	Trocar para o ponteiro da pilha da nova tarefa;	2-50
7	Restaurar o contexto da nova tarefa;	20-100
8	Retornar para a nova tarefa a partir da localização em que se encontrava o seu PC anterior;	2-40
9	Outro processameto.	0-100
Total		50-500

Este tempo é, segundo (LI e YAO, 2003), relativamente insignificante comparado à maioria das operações que uma tarefa executa. Entretanto, se uma aplicação possui uma freqüente troca de contexto, pode apresentar um “overhead” desnecessário, diminuindo muito seu desempenho. A idéia é desenvolver aplicações que não envolvam muitas trocas de contexto.

Segundo, (OSHANA, 2007), o escalonador é executado na mesma CPU que as tarefas do desenvolvedor da aplicação, o que já traz uma penalidade para o sistema na utilização dos serviços do escalonador. Existem três motivos para o kernel tomar o controle da CPU da tarefa em execução e executar a si mesmo:

- Responder a uma chamada de sistema.
- Realizar o escalonamento e os serviços de tempo;
- E lidar com as interrupções externas.

5.5.Algoritmos de escalonamento

O escalonador determina qual tarefa será executada pela CPU de acordo com o algoritmo de escalonamento (ou política de escalonamento). A maioria dos kernels de hoje possuem dois tipos de algoritmos: o “preemptivo-prioritário” e o “round-robin”.

5.5.1.Escalonamento preemptivo-prioritário

Neste tipo de escalonamento, a tarefa de maior prioridade, entre todas as tarefas do sistema que estiverem prontas para execução, será a tarefa que tomará, imediatamente, o controle da CPU. A Figura 8 ilustra esta forma de escalonamento.

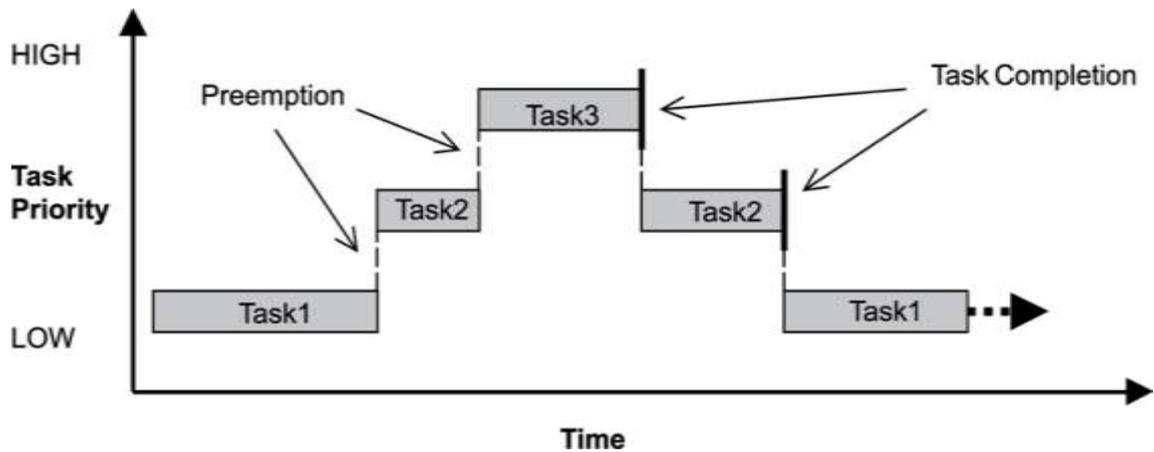



Figura 8. Escalonamento preemptivo-prioritário. (LI e YAO, 2003).

Segundo (LI e YAO, 2003), com um escalonamento preemptivo-prioritário, cada tarefa tem sua prioridade e aquela de maior prioridade é a primeira a ser executada. Se uma tarefa com a prioridade mais alta fica pronta para ser executada, o kernel salva, imediatamente, o contexto da tarefa em execução no TCB da mesma, e libera a CPU para a tarefa de mais alta prioridade ser executada. Como mostra a Figura 8, a tarefa 1 sofre preempção da tarefa 2, pois esta possui uma prioridade maior. A tarefa 2 sofre então preempção da tarefa 3, por esta ter a mais alta prioridade no momento. Quando a tarefa 3 completa sua execução, a tarefa 2 retorna à sua execução. Quando esta última acaba sua execução, é então a vez da tarefa 1 ser executada.

Alguns kernels de tempo real suportam até 256 níveis de prioridade, sendo 0 o de mais alta prioridade e 255 o de mais baixa. Alguns tratam a prioridade na ordem inversa, com a prioridade 255 como a mais alta e a 0 como a mais baixa, entretanto o principio é o mesmo.

Apesar de algumas tarefas terem suas prioridades determinadas quando são criadas, a prioridade de uma tarefa pode ser trocada dinamicamente usando chamadas fornecidas pelo kernel. Esta habilidade permite que uma aplicação

embarcada possua flexibilidade para se ajustar a eventos externos à medida que eles ocorrem, criando um verdadeiro sistema com resposta em tempo real. Mas esta é uma ferramenta que deve ser usada com cuidado, pois pode levar o sistema a problemas de inversão de prioridade, deadlock e eventuais falhas.

5.5.2.Escalonamento Round-robin

De acordo com (LI e YAO, 2003), o escalonamento Round-robin faz com que o tempo de execução da CPU seja dividido igualmente entre as tarefas. Um puro escalonamento round-robin não consegue satisfazer os requisitos de um sistema de tempo real porque, nestes sistemas, as tarefas possuem vários graus de importância. No entanto, um escalonamento round-robin pode ser adicionado a um escalonamento preemptivo-prioritário para dividir as fatias de tempo da CPU igualmente entre as tarefas que possuem a mesma prioridade, conforme representação na Figura 9.

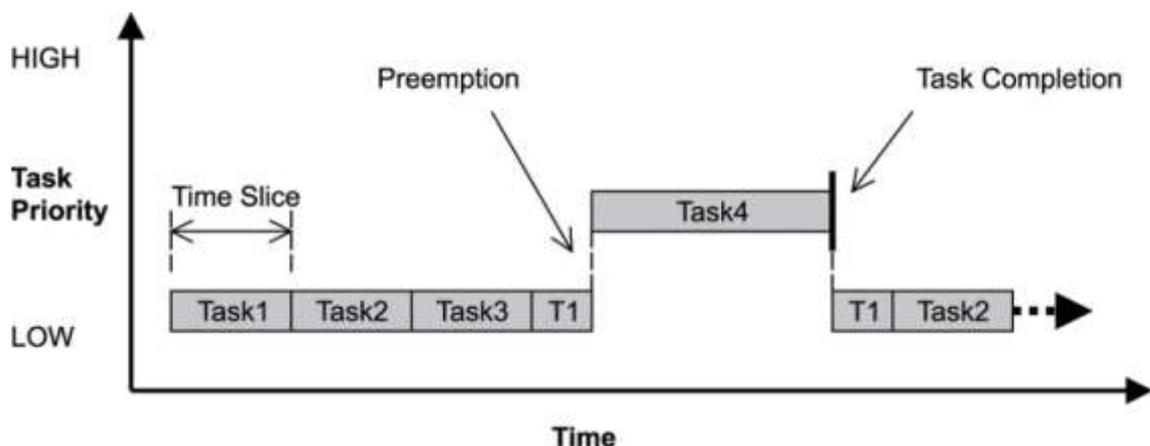



Figura 9. Escalonamento round-robin. (LI e YAO, 2003).

Com a divisão do tempo em fatias iguais, cada tarefa é executa por este intervalo de tempo definido, em um ciclo contínuo, o qual é o *round robin*. Um contador do tempo de execução conta a fatia de tempo de cada tarefa, sendo incrementado a cada *tick* de *clock.* Quando a fatia de tempo de uma tarefa termina, o contador é zerado e a tarefa é colocada no final do ciclo. Tarefas de mesma prioridade, à medida que são adicionadas na lista de tarefas prontas, são colocadas no final do ciclo, com seus contadores de tempo de execução inicializados com zero.

Se uma tarefa sofre preempção de uma tarefa de maior prioridade, em um ciclo round robin, seu contador de tempo de execução é salvo e, então, restaurado quando a tarefa interrompida está novamente pronta para ser executada. A Figura 9 ilustra essa idéia ao mostrar a tarefa 1 sofrendo preempção da tarefa 4 (de maior prioridade). Quando a tarefa 4 é executada por completo, a tarefa 1 volta a ser executada, do instante em que parou.

5.6.Objetos do kernel

Os objetos do kernel são, segundo (LI e YAO, 2003), as principais unidades no desenvolvimento de sistemas embarcados de tempo real. Os objetos mais comuns são:

- **Tarefas** – são funções independentes e concorrentes entre si que competem pelo tempo de execução da CPU.
- **Semáforos** – são sinalizadores utilizados para sincronização ou exclusão mútua, podendo ser incrementados ou decrementados por uma tarefa.

- **Filas de mensagens** – são estruturas de dados do tipo buffer[3] que podem ser usadas para sincronização, exclusão mútua e troca de dados entre as tarefas.

Existem outros tipos de objetos do kernel para auxiliar na resolução de problemas de tempo real como, por exemplo, caixas de correio, pipes, flags, mutexes, contadores e etc. No entanto, os três objetos básicos aqui citados resolvem os problemas mais comuns que são a concorrência, a sincronização e a transferência de dados entre as tarefas.

5.6.1.Tarefas

Uma tarefa é, conforme (LI e YAO, 2003), uma "função" independente do programa que pode concorrer com outras "funções", também independentes, pelo tempo de execução do processador. Utilizando este conceito, os desenvolvedores podem decompor seu aplicativo em múltiplas tarefas para otimizar o tratamento das entradas e saídas do sistema, atendendo assim, seus requisitos de tempo.Uma tarefa é a unidade básica de programação que os sistemas operacionais controlam. De acordo com (GANSSLE, 1998), o kernel cria a tarefa, aloca um espaço de memória para a mesma e grava o código a ser executado pela tarefa na memória.

Segundo (LI e YAO, 2003), uma tarefa possui um conjunto de parâmetros e estruturas de dados que são associados a ela na sua criação:

- Um Nome;
- Um ID único;

[3] Área da memória do computador reservada para armazenar dados temporariamente.

- Uma prioridade (se for utilizado um escalonamento preemptivo-priorirátio);
- Um bloco de controle da tarefa (TCB) que contem as informações necessárias para o escalonamento da tarefa;
- Uma pilha;
- Uma rotina de execução.

São estes componentes que fazem com que as tarefas sejam reconhecidas como objetos tarefas e diferenciadas das funções com que se trabalha em aplicações que não utilizem sistemas operacionais. A Figura 10 ilustra uma tarefa e seus componentes.

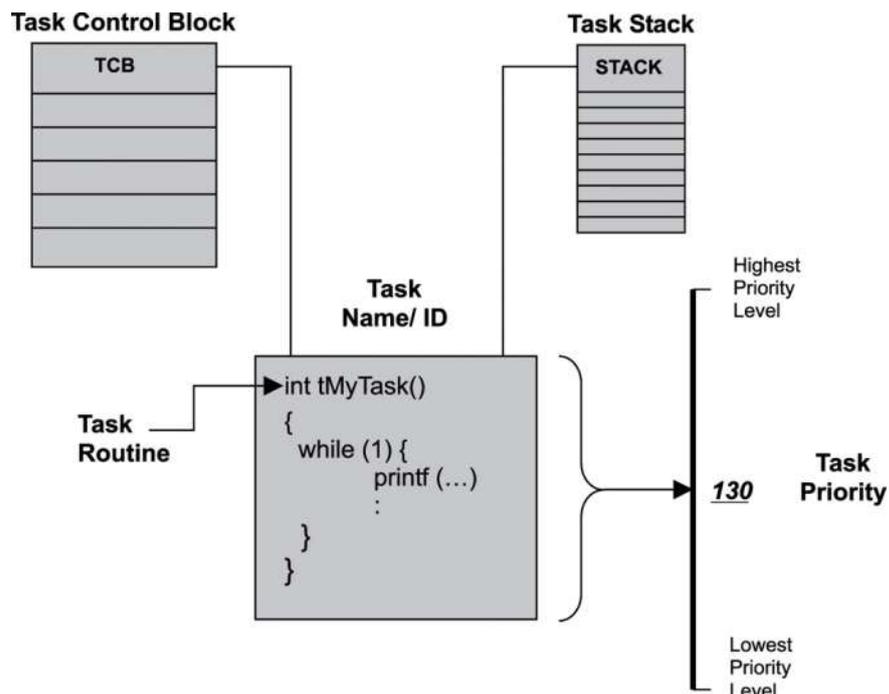



Figura 10. Representação ilustrativa de uma tarefa. (LI e YAO, 2003).

Durante sua inicialização, o kernel cria um grupo de tarefas, denominadas de tarefas de sistema, e aloca prioridades apropriadas para cada. Estas prioridades são reservadas pelo RTOS para que ele as aplique a estes tipos específicos de tarefas.

As prioridades das tarefas da aplicação não devem ter o mesmo nível de prioridade das tarefas do sistema, podendo afetar o desempenho e o comportamento do mesmo, caso isto aconteça. O kernel precisa das suas tarefas de sistema e de suas prioridades reservadas para operar corretamente.

Após o kernel ter sido inicializado e ter criado todas as tarefas de sistema que ele necessita, ele passa a executar um ponto de entrada pré-definido (como, por exemplo, uma função pré-definida) que serve para inicializar a aplicação. Deste ponto de entrada, o desenvolvedor pode inicializar e criar outras tarefas da aplicação, bem como os objetos do kernel que sua aplicação necessita. À medida que o desenvolvedor cria novas tarefas ele deve determinar qual o nome da tarefa, a prioridade e a rotina da mesma. O kernel então determina uma ID única, cria um TCB associado e um espaço de memória para cada tarefa criada pelo desenvolvedor da aplicação.

5.6.1.1.Estado das tarefas

De acordo com (LI e YAO, 2003), uma tarefa criada pelo kernel sempre se encontra em um estado específico, que será determinante na escolha das tarefas a serem executadas. Apesar de os kernels poderem definir estados diferentes para as tarefas, geralmente três principais estados são definidos na maioria dos kernels preemptivos-prioritários:

- **Estado "pronto"** – estado em que uma tarefa está pronta para execução, mas não está sendo executada porque uma tarefa de maior prioridade possui a CPU.

- **Estado "bloqueado"** – estado em que uma tarefa requisita um recurso que não está disponível, espera por algum evento ocorrer ou atrasou sua execução por uma determinada duração.
- **Estado "em execução"** – estado em que a tarefa está em execução por ter a maior prioridade no momento.

À medida que a aplicação está sendo executada, cada tarefa muda-se de um estado para outro, de acordo com a lógica de uma simples máquina de estado finita (FSM – finite state machine). A Figura 11 ilustra uma típica FSM para os estados das tarefas, com um breve comentário sobre as transições dos estados.

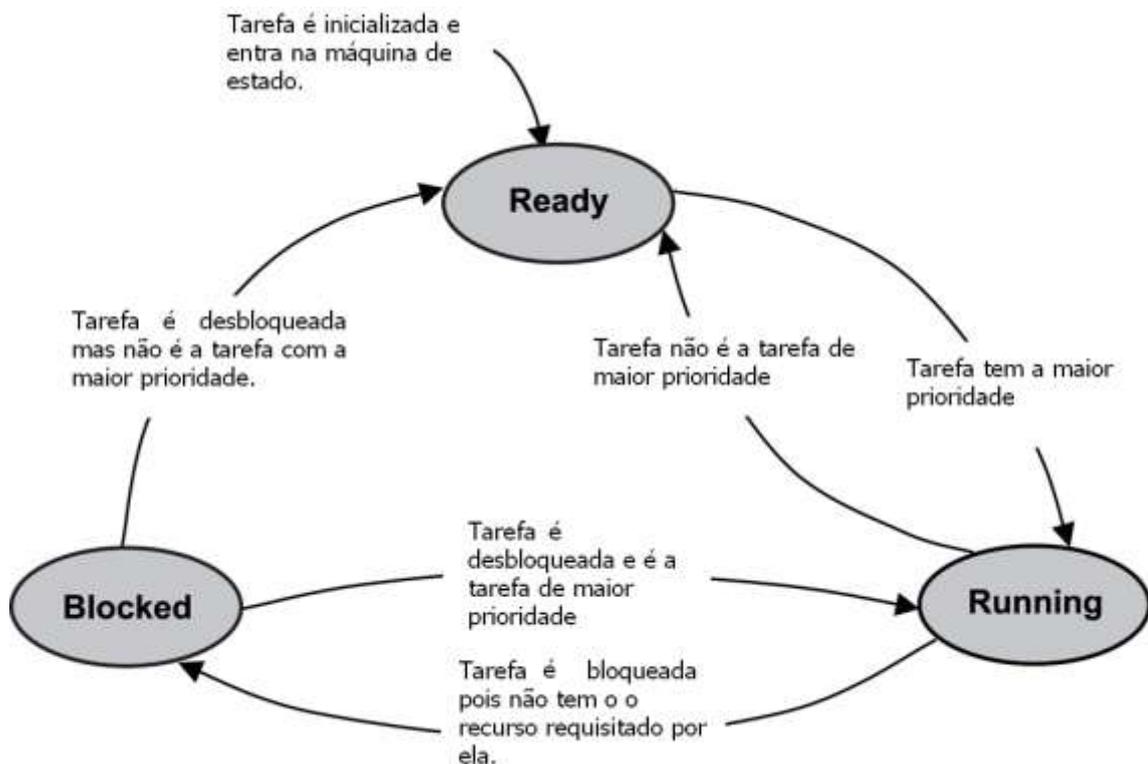



Figura 11. Máquina de estado das tarefas. (LI e YAO, 2003).

O kernel deve manter os estados em que se encontram todas as tarefas de um sistema. Este estado é mantido devido aos TCBs. Eles guardam as informações necessárias para manter o estado da tarefa quando ela sofre uma preempção e

permite que ela, após ganhar o controle da CPU novamente, continue sua execução do ponto onde sofreu a preempção. Conforme as tarefas em execução fazem chamadas de sistemas ao kernel para utilizar seus serviços, o escalonador deve determinar qual tarefa precisa mudar de estado e então faz a troca do mesmo.

É importante observar que uma mudança de estado não implica em uma mudança de contexto. Este último só ocorrerá quando a tarefa de maior prioridade do sistema está pronta para se executada.

5.6.1.1.1.O estado “pronto”

Conforme (LI e YAO, 2003), quando uma tarefa é criada e fica pronta para a execução, o kernel coloca esta tarefa no estado “pronto”. Neste estado, a tarefa compete ativamente com todas as outras tarefas pelo tempo de execução do processador. Conforme ilustra a Figura 11, uma tarefa no estado “pronto” não pode mudar-se diretamente para o estado “bloqueado”, pois necessita ser executada para poder fazer uma chamada (denominada de chamada de bloqueio) que possa colocá-la num estado “bloqueado”. Tarefas “prontas” só podem, portanto, mudarem-se para o estado “em execução”. Como muitas tarefas podem estar simultaneamente no estado “pronto”, o escalonador usa a prioridade de cada tarefa para determinar qual tarefa irá mudar para o estado “em execução”.

5.6.1.1.2.O estado “em execução”

De acordo com (LI e YAO, 2003), quando uma tarefa muda para o estado “em execução”, o processador carrega seus registradores com o conteúdo daquela tarefa. Dessa forma, o processador pode executar as instruções da tarefa e manipular sua pilha associada.

Conforme ilustra a Figura 11, uma tarefa pode mudar do estado “em execução” para o estado “pronto” quando sofre preempção de uma tarefa de maior prioridade. Neste caso, a tarefa que sofreu a preempção é colocada na lista de tarefas pronta, enquanto a tarefa de maior prioridade é removida desta lista ao trocar do estado “pronto” para o estado “em execução”.

Uma tarefa em execução pode mudar para o estado “bloqueado” das seguintes maneiras:

- Fazendo uma chamada de sistema para pedir um recurso não disponível;
- Fazendo uma chamada de sistema para esperar que um evento ocorra;
- Fazendo uma chamada de sistema para se atrasar por um determinado período.

5.6.1.1.3.O estado “bloqueado”

Segundo (LI e YAO, 2003), o estado “bloqueado” é extremamente importante em sistemas de tempo real porque permitem que tarefas de menor prioridade possam obter o controle da CPU. Se as tarefas com as mais altas prioridades não entrarem no estado bloqueado, pode ocorrer o que se chama de *starvation*, ou seja, as tarefas de maior prioridade usam todo o tempo de execução da CPU, não permitindo que as tarefas de menor prioridade sejam executadas.

Uma tarefa só pode mudar para o estado “bloqueado” fazendo uma chamada de bloqueio, ou seja, requisitando que alguma condição de bloqueio seja atendida. Uma tarefa continua bloqueada até que a condição de bloqueio é atendida. Alguns exemplos de quando estas condições de bloqueio são atendidas são:

- Um sinalizador do semáforo que uma tarefa está esperando é liberado;
- Uma mensagem que a tarefa está esperando chega à caixa de correio;
- O tempo de atraso de uma tarefa expira.

Ao ser desbloqueada, uma tarefa muda do estado “bloqueado” para o estado “pronto” se ela não for a tarefa de maior prioridade. A tarefa entra, portanto, na lista de tarefas prontas na colocação apropriada a usa prioridade. Quando uma tarefa é desbloqueada e é a de maior prioridade naquele momento, ela muda diretamente para o estado “em execução” (sem passar pelo estado “pronto”), tomando a CPU da tarefa que estiver em execução e enviando esta para a fila de tarefas prontas.

5.6.1.2.Estruturas típicas das tarefas

Os códigos das tarefas podem ser estruturados de duas formas:

5.6.1.2.1.“Execução até o fim”

A estrutura de código do tipo “execução até o fim” é mais utilizado para inicializações. Geralmente, as tarefas executam uma só vez, quando o sistema é ligado, inicializando o aplicativo e criando os serviços, objetos e tarefas adicionais necessários para a aplicação.

As tarefas que iniciam os aplicativos geralmente têm maior prioridade que as tarefas criadas por elas para que seu trabalho de inicialização não sofra preempção. Após completar seu trabalho, as tarefas de inicialização são suspensas ou apagam a si mesmas para que as tarefas criadas por ela, com menores prioridades possam ser executadas.

O pseudo-código deste tipo de tarefa é mostrado a seguir:

```
RunToCompletionTask ()
{
    Initialize application
    Create ‘endless loop tasks'
    Create kernel objects
    Delete or suspend this task
}
```

5.6.1.2.2.“Loop sem fim”

As tarefas do tipo “loop sem fim” fazem a maior parte do trabalho do aplicativo, tratando das entradas e saídas. Elas rodam várias vezes enquanto o sistema está ligado. Estas tarefas podem conter, também, códigos de inicialização. No entanto, ele só será executado na primeira vez que a tarefa for executada, pois a tarefa ficará executando o loop sem fim. É no corpo do loop que serão feitas as chamadas de bloqueio para permitir que tarefas de menor prioridade sejam executadas.

O pseudo-código para as tarefas de “loop sem fim” é apresentado a seguir:

```
EndlessLoopTask ()
{
    Initialization code
    Loop Forever
    {
        Body of loop
        Make one or more blocking calls
    }
```

}

5.6.2.Semáforos

Um semáforo, ou um sinalizador de semáforo, é, de acordo com (LI e YAO, 2003), um objeto do kernel que uma ou mais tarefas pode adquirir ou liberar para fins de sincronização, exclusão mútua ou sinalização da ocorrência de um evento. Quando criado, o kernel associa ao semáforo um bloco de controle, um nome, uma ID única, um valor e uma lista de tarefas-em-espera, conforme ilustra a Figura 12.

Uma maneira mais simples de se entender o conceito de semáforo é imaginá-lo como uma chave que permite a conclusão de uma operação ou o acesso a um recurso. Se uma tarefa puder adquirir esta chave (ou sinalizador de semáforo), ela poderá concluir sua operação ou acessar um recurso.

Um único semáforo pode ser adquirido finitas vezes. No entanto, quando este limite de vezes é alcançado, nenhuma tarefa pode adquirir o semáforo, até que alguma outra tarefa libere-o novamente.

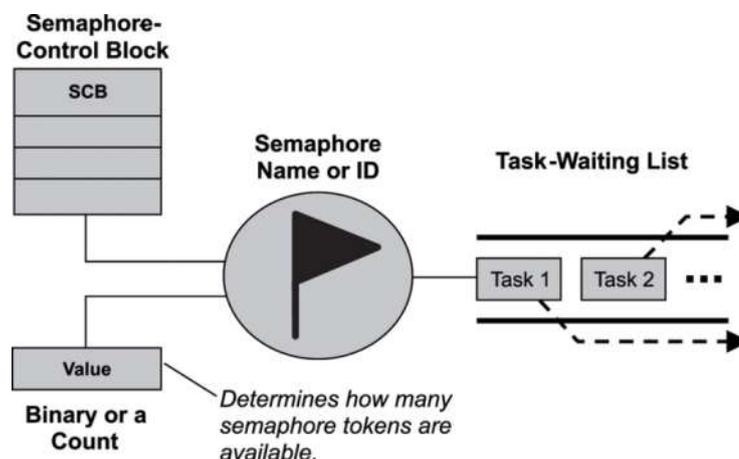



Figura 12. Representação ilustrativa de um semáforo. (LI e YAO, 2003).

O kernel rastreia o número de vezes que o semáforo foi adquirido ou liberado através de um contador (de chaves ou sinalizadores), que é inicializado com um valor especificado na criação de um semáforo. Quando uma tarefa adquire o semáforo, o contador é decrementado e quando uma tarefa libera o semáforo, o contador é incrementado. Se o contador alcança o valor zero, o semáforo não possui nenhuma chave, logo a tarefa requisitante não poderá adquirir o semáforo e poderá ficar bloqueada, se esta escolher esperar pela disponibilidade do semáforo. Todas as tarefas bloqueadas por um semáforo indisponível são mantidas em uma lista de tarefas-em-espera que podem estar ordenadas tipo *first in/ first out* (FIFO) ou então as maiores prioridades primeiro.

Quando um semáforo fica disponível, o kernel permite que a primeira tarefa da lista de tarefas-em-espera adquira o semáforo (ou sua chave), mudando o estado desta tarefa para o estado “em execução”, caso esta tarefa seja a de maior prioridade, ou então para o estado “pronto”, até que a tarefa se torne a de maior prioridade e possa ser executada.

Os tipos mais básicos de semáforos que um kernel geralmente apresenta são o semáforo binário e o semáforo contador.

5.6.2.1.O semáforo binário

Segundo (LI e YAO, 2003), um semáforo binário pode ter somente os valores 0 ou 1. Quando um semáforo binário possui o valor 0, o semáforo está vazio e indisponível; quando este valor é 1, o semáforo binário é considerado cheio e disponível. Na criação de um semáforo binário, pode-se inicializá-lo para estar

disponível ou indisponível (1 ou 0, respectivamente) para o seu uso inicial. O diagrama de estados dos semáforos binários é mostrado na Figura 13.

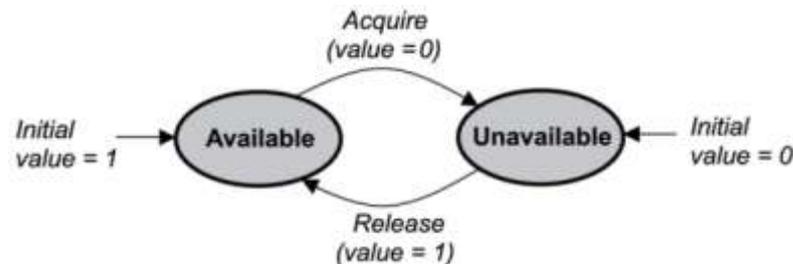



Figura 13. Diagrama de estados de um semáforo binário. (LI e YAO, 2003).

Semáforos binários são tratados como recursos globais, o que significa que eles são compartilhados entre todas as tarefas que necessitam deles. Fazer de um semáforo um recurso global permite que qualquer tarefa o libere, mesmo se a tarefa não o adquiriu inicialmente.

5.6.2.2.Semáforos contadores

Um semáforo contador, como o nome já diz, possui um contador que permite que este tipo de semáforo possa ser adquirido e liberado várias vezes. Conforme (LI e YAO, 2003), quando um semáforo contador é criado, ele é inicializado com um número especificado de sinalizadores, que será o valor máximo de seu contador. Se este valor é inicializado com 0, o semáforo é criado no estado indisponível. Se o valor é criado com um valor maior que 0, o semáforo é criado no estado disponível. O diagrama de estado dos semáforos contadores é apresentado na Figura 14.

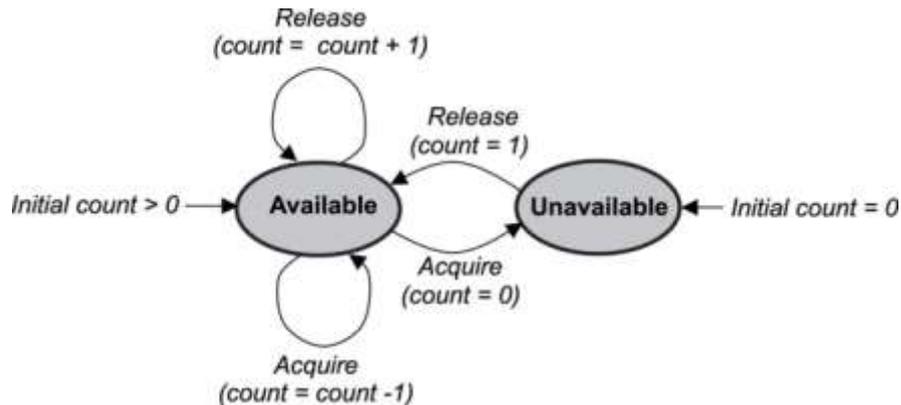



Figura 14. Diagrama de estados de um semáforo contador. (LI e YAO, 2003).

Tarefas podem continuamente adquirir sinalizadores de um semáforo contador até este não possuir mais nenhum sinalizador. Quando todos os sinalizadores acabarem, o contador é igualado a 0 e o semáforo contador muda do estado de disponível para o estado indisponível. Para o semáforo voltar ao estado disponível, um sinalizador do mesmo deve ser liberado por alguma tarefa.

O semáforo contador é um recurso global, assim como o semáforo binário.

5.6.3.Filas de mensagens

Uma fila de mensagens é, segundo (LI e YAO, 2003), um objeto do tipo buffer, através do qual, tarefas ou ISRs enviam e recebem mensagens para comunicação e sincronia com dados. A fila armazena temporariamente uma mensagem de um remetente até que o destinatário esteja pronto para recebê-la. O uso deste buffer temporário separa a tarefa remetente da tarefa destinatária, ou seja, ele libera as tarefas de ter que enviar e receber as mensagens simultaneamente.

Como os objetos do kernel citados anteriormente, uma fila de mensagens tem também uma série de componentes associados a ela que auxiliam ao kernel

gerenciá-la. Quando uma fila de mensagens é criada, é associado a ela um bloco de controle da fila (QCB – queue control block), um nome, uma ID única, buffers de memória, um tamanho para fila, um tamanho para as mensagens da fila, e uma ou mais listas de tarefas-em-espera, conforme ilustra a Figura 15.

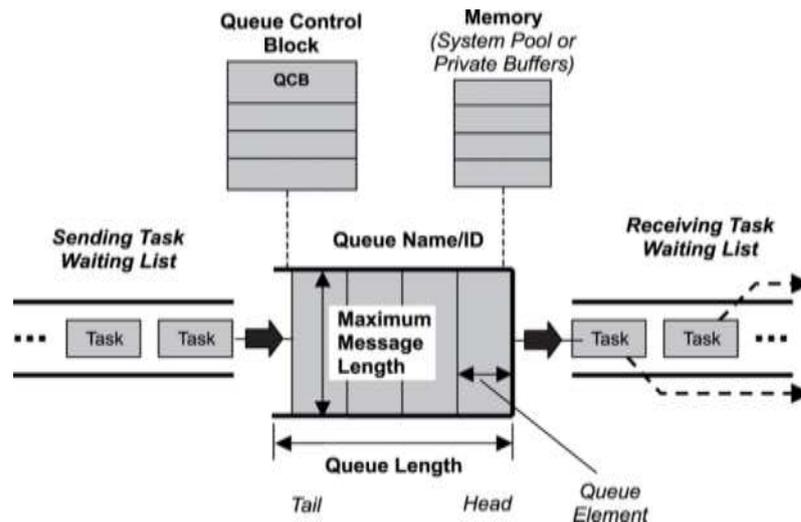



Figura 15. Ilustração de uma fila de mensagens e seus componentes associados. (LI e YAO, 2003).

É função do kernel associar uma única ID para a fila de mensagens e criar seu bloco de controle (QCB). O kernel também determina o quanto de memória é necessário para a fila de mensagens através dos parâmetros especificados pelo desenvolvedor, depois que este determina o tamanho da fila e o tamanho máximo das mensagens. Após ter essa informação, o kernel aloca a memória necessária para a fila de mensagem em uma única e grande área de memória compartilhada entre todas as filas de mensagens, ou então em áreas de memória separadas e individuais para cada fila de mensagens.

Uma fila de mensagens, propriamente dita, consiste de um número de elementos, os quais guardam, individualmente, uma mensagem única. Os elementos

que guardam a primeira e a última mensagem são chamados de cabeça e rabo, respectivamente. Alguns elementos da fila podem estar vazios (sem nenhuma mensagem), e o número total de elementos da fila (vazios ou não) determina o comprimento total da mesma.

Conforme mostra a Figura 15, uma fila de mensagens pode ter duas filas de tarefas-em-espera: a lista de espera das tarefas destinatárias, que consiste das tarefas que esperam receber as mensagens da fila de mensagens quando ela está vazia; e a lista de espera das tarefas remetentes, que são as tarefas que estão esperando um lugar para depositar uma mensagem na fila de mensagens quando ela está cheia.

5.6.3.1.Os estados das filas de mensagens

De acordo com (LI e YAO, 2003), as filas de mensagens seguem a lógica das máquinas de estados finitas (FSMs) conforme mostra a Figura 16. Quando uma fila de mensagens é criada, a máquina de estado se encontra no estado “vazio”. Se uma tarefa espera receber uma mensagem da fila de mensagens enquanto ela está vazia, a tarefa entra no estado “bloqueado”, e se esta quiser permanecer neste estado, ela é armazenada na lista de tarefas-em-espera associada a aquela fila de mensagens. A ordem das tarefas na lista de tarefas-em-espera de uma fila de mensagens é do tipo FIFO ou de acordo com sua prioridade. Se outra tarefa envia uma mensagem para a fila de mensagens, a mensagem é entregue diretamente para aquela tarefa bloqueada. A tarefa bloqueada é então removida da lista de tarefas-em-espera e seu estado é trocado para o estado “pronto” ou “em execução”.

Neste caso, a fila de mensagens continua vazia, pois a mensagem foi enviada com sucesso.

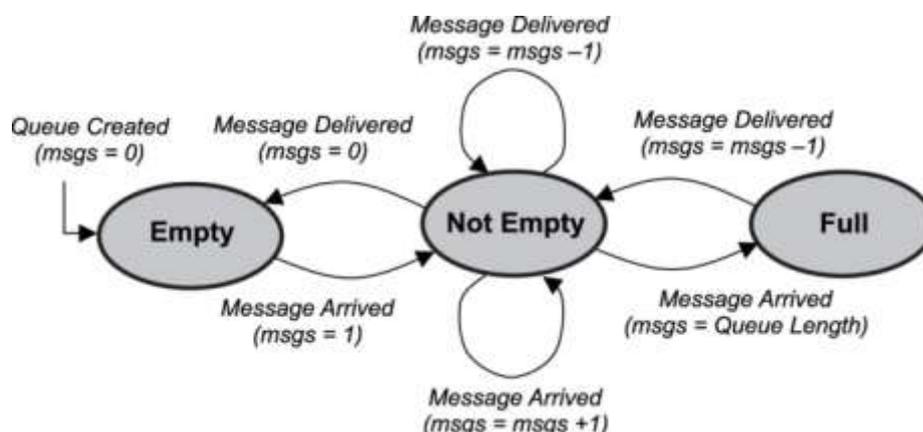



Figura 16. FSM de uma fila de mensagens. (LI e YAO, 2003).

Continuando neste mesmo cenário, se outra mensagem é enviada para a mesma fila de mensagens e não há nenhuma tarefa na lista de tarefas-em-espera para recebê-la, a fila de mensagens muda seu estado para “não vazio”. À medida que mensagens chegam à fila, esta vai se enchendo até que seu espaço livre é esgotado. Neste ponto, o número de mensagens é igual ao tamanho da fila, e a fila de mensagens troca seu estado para “cheio”. Enquanto a fila de mensagens está neste estado, nenhuma tarefa que tentar enviar uma mensagem à fila terá sucesso, pelo menos até outra tarefa requisitar uma mensagem da fila de mensagens, liberando um elemento da mesma.

Em alguns kernels, quando uma tarefa tenta enviar uma mensagem para uma fila de mensagens que está cheia, a função de envio retorna uma mensagem de erro para a tarefa. Em outros kernels, a tarefa pode ser bloqueada e enviada para a lista de tarefas-em-espera das tarefas remetentes, que é uma fila de espera separada das tarefas destinatárias.

5.6.3.2.O conteúdo das filas de mensagens

Conforme (LI e YAO, 2003), filas de mensagens podem ser usadas para enviar e receber uma variedade de dados, como por exemplo, o valor de temperatura de um sensor, um texto a ser impresso no LCD, um evento do teclado e etc. Algumas destas mensagens podem ser muito longas e exceder o tamanho máximo da mensagem, que é determinado na criação da fila. Uma maneira de resolver este problema de limitação do tamanho da mensagem é enviar ponteiros para os dados, ao invés dos dados propriamente ditos. Mesmo se uma mensagem longa couber na fila, é melhor mandar um ponteiro para aquela mensagem, pois melhora o desempenho e a utilização de memória.

Ao ser enviada de uma tarefa para outra, a mensagem é normalmente copiada duas vezes. A primeira cópia acontece quando a mensagem é copiada da área de memória da tarefa remetente para a área de memória da fila de mensagem. A segunda cópia ocorre quando a mensagem é copiada da área de memória da fila de mensagem para a área de memória da tarefa destinatária. A Figura 17 ilustra esse cenário.

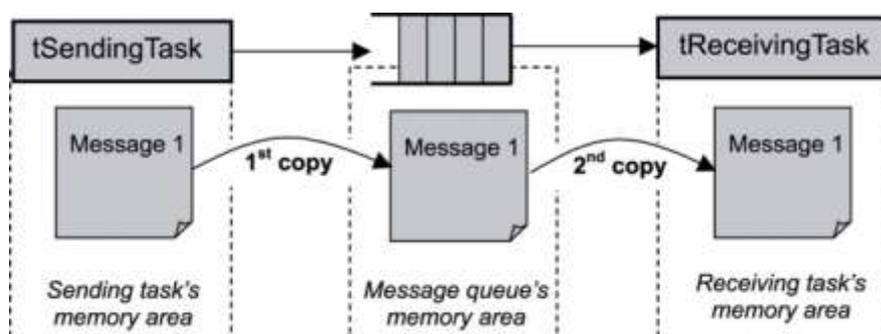



Figura 17. Cópia de mensagens entre as tarefas e a fila de mensagens. (LI e YAO, 2003).

Uma exceção pode ocorrer quando a tarefa destinatária está bloqueada, esperando pela mensagem na lista de tarefas-em-espera da fila de mensagens. Dependendo da implementação do kernel, a mensagem pode ser copiada apenas uma vez, da área de memória da tarefa remetente, diretamente para a área de memória da tarefa destinatária, não precisando passar pela área de memória da fila de mensagens.

Devido a copia de dados ser custosa em termos de desempenho e requisitos de memória, é interessante, no tratamento de sistemas embarcados de tempo real, sempre trabalhar com mensagens pequenas ou então utilizar ponteiros.

5.6.3.3.Caixas de correio

Uma fila de mensagens cujo tamanho é de uma única mensagem determina um novo objeto do kernel, denominado de caixa de correio. Caixas de correio possuem as mesmas características e funcionalidades que as filas de mensagens, sendo que o que as diferenciam são suas máquinas de estados. A máquina de estado de uma caixa de correio é representada na Figura 18.

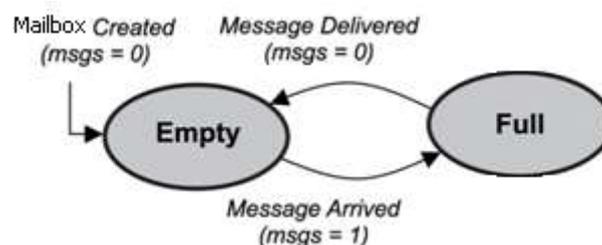



Figura 18. FSM de uma caixa de correio.

5.7.Serviços do kernel

Junto com os objetos, a maioria dos kernels fornece serviços ou operações que auxiliam os desenvolvedores a criar aplicações para sistemas embarcados de tempo real. Estes serviços são, conforme (LI e YAO, 2003), um conjunto de chamadas APIs[4] utilizadas para executar operações sobre os objetos do kernel, facilitar o gerenciamento do tempo, o tratamento das interrupções, dos dispositivos de entrada e saída, e o gerenciamento da memória.

Os serviços aqui descritos serão os relacionados aos objetos citados na seção anterior. Os nomes destes serviços são diferentes entre um kernel e outro, mas apesar da diferença de nome, eles possuem a mesma funcionalidade. Aqui serão descritos nomes genéricos de acordo com a operação.

5.7.1.Serviços de gerenciamento das tarefas

Os serviços de gerenciamento das tarefas incluem tanto as chamadas APIs, quanto as ações que o kernel executa “nos bastidores” para auxiliar no gerenciamento das tarefas, como por exemplo, a criação e manutenção dos TCBs e as pilhas das tarefas.

---

[4] API, de Application Programming Interface (ou Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por outros programas, isto é: programas que não querem envolver-se em detalhes da implementação de um software, mas apenas usar seus serviços. De modo geral, a API é composta por uma série de funções acessíveis somente por programação, e que permitem utilizar características de um software que são menos evidentes ao programador.

5.7.1.1.Criando e apagando tarefas

A Tabela 2 mostra as duas operações fundamentais que um desenvolvedor deve aprender para manipular tarefas: criar e apagar tarefas.

Tabela 2. Operações para criar e apagar tarefas. (LI e YAO, 2003).

Operação	*Descrição da operação*
Create	Cria uma tarefa.
Delete	Apaga uma tarefa.

Segundo (LI e YAO, 2003), geralmente cria-se uma tarefa usando uma ou duas operações, dependendo da chamada API do kernel. Alguns kernels permitem ao desenvolvedor criar uma tarefa e posteriormente iniciá-la. Neste caso a tarefa é criada e colocada num estado “suspenso”, onde não pode ser executada. A tarefa só poderá ser executada a partir do momento em que é inicializada pelo desenvolvedor. O estado “suspenso” é similar ao estado “bloqueado” no sentido de que a tarefa não está nem “em execução” nem “pronta” para a execução, possuindo algumas diferenças. Estas diferenças não serão abordadas aqui, pois variam muito entre as dezenas de kernels existentes.

Na maioria dos casos, os kernels criam e inicializam uma tarefa em uma única operação. É importante observar que inicializar uma tarefa não faz com que esta seja executada imediatamente, mas a coloca na lista de tarefas-prontas.

Muitas implementações de kernel permitem que qualquer tarefa apague qualquer outra tarefa. Durante este procedimento, o kernel elimina a tarefa e libera a memória ocupada por ela, apagando seu TCB e sua pilha.

Entretanto, é importante ter cuidado ao se apagar uma tarefa, pois durante sua execução, ela pode adquirir memória ou acessar recursos utilizando outros objetos do kernel. Se isto ocorrer e a tarefa for apagada incorretamente, ela pode não conseguir liberar estes recursos. Imagine por exemplo que uma tarefa adquiriu um sinalizador (chave) de um semáforo, ganhando acesso exclusivo a uma estrutura de dados compartilhada. Enquanto a tarefa está operando sobre a estrutura de dados, a tarefa é apagada. Se não for tratado corretamente, o apagamento abrupto da tarefa pode resultar em:

- Uma estrutura de dados corrompida devido a uma operação incompleta de escrita;
- Um semáforo inacessível para outras tarefas que necessitem adquiri-lo;
- Uma estrutura de dado inacessível, devido a um semáforo inacessível

Apagar uma tarefa prematuramente pode resultar em vazamento de memória ou de recurso. Um vazamento de memória ocorre quando uma quantidade de memória é adquirida e não é liberada, o que pode, eventualmente, deixar o sistema sem memória para ser executado corretamente. Um vazamento de recurso ocorre quando um recurso é adquirido e nunca é liberado, resultando em um vazamento de memória, pois recursos ocupam espaço na memória.

5.7.1.2.Escalonando manualmente tarefas

Apesar das tarefas mudarem de estados automaticamente, como resultado da execução do programa e do escalonamento, muitos kernels fornecem um conjunto de chamadas APIs que permitem aos desenvolvedores controlar quando uma tarefa muda para um estado diferente. Esta capacidade é denominada de escalonamento manual e suas principais operações são listadas na Tabela 3.

Tabela 3. Operação para escalonamento manual das tarefas. (LI e YAO, 2003).

Operação	*Descrição da operação*
Suspend	Suspende a execução uma tarefa.
Resume	Executa a tarefa a partir do ponto onde ela foi suspensa.
Delay	Atrasa a execução de uma tarefa.
Restart	Reinicia a execução de uma tarefa.
Get Priority	Obtém a prioridade da tarefa requisitante.
Set Priority	Configura dinamicamente a prioridade da tarefa requisitante.
Preemption lock	Impede que a tarefa requisitante sofra preempção de tarefas com maior prioridade.
Preemption unlock	Faz com que a tarefa requisitante volte a sofrer preempção de tarefas com maior prioridade.

De acordo com (LI e YAO, 2003), o escalonamento manual permite aos desenvolvedores suspender ou resumir a execução de uma tarefa através do aplicativo. Permite também atrasar uma tarefa para, por exemplo, esperar por uma condição externa que não está associada a uma interrupção. Atrasar uma tarefa significa retirar a tarefa da CPU (mantendo-a bloqueada pela quantidade de tempo determinada) e permitir que outra tarefa seja executada. Depois que o tempo de

atraso expirar, a tarefa que foi retirada da CPU retorna para a lista de tarefas-prontas.

O desenvolvedor do aplicativo pode também querer reiniciar uma tarefa, o que é diferente de resumir uma tarefa que foi suspensa. Reiniciar uma tarefa significa executá-la do início como se ela não tivesse sido executada anteriormente. Todo o estado interno (os registradores da CPU e os recursos adquiridos) que a tarefa mantinha quando foi suspensa é perdido quando esta é reiniciada. Na operação de resumo de uma tarefa, o estado interno é mantido para que a tarefa seja executada do mesmo ponto onde foi suspensa.

As operações “get priority” e “set priority” são importantes para resolver problemas de inversão de prioridades. Uma inversão de prioridades ocorre quando uma tarefa de menor prioridade possui um recurso compartilhado[5] requisitado por uma tarefa de alta prioridade. No entanto, a tarefa de menor prioridade sofreu preempção de uma tarefa de média-prioridade. A Figura 19 ilustra este cenário.

---

[5] Um recurso compartilhado é qualquer entidade que pode ser usada por mais de uma tarefa, sendo que cada tarefa deve ganhar acesso exclusivo ao recurso compartilhado para evitar que os dados sejam corrompidos.

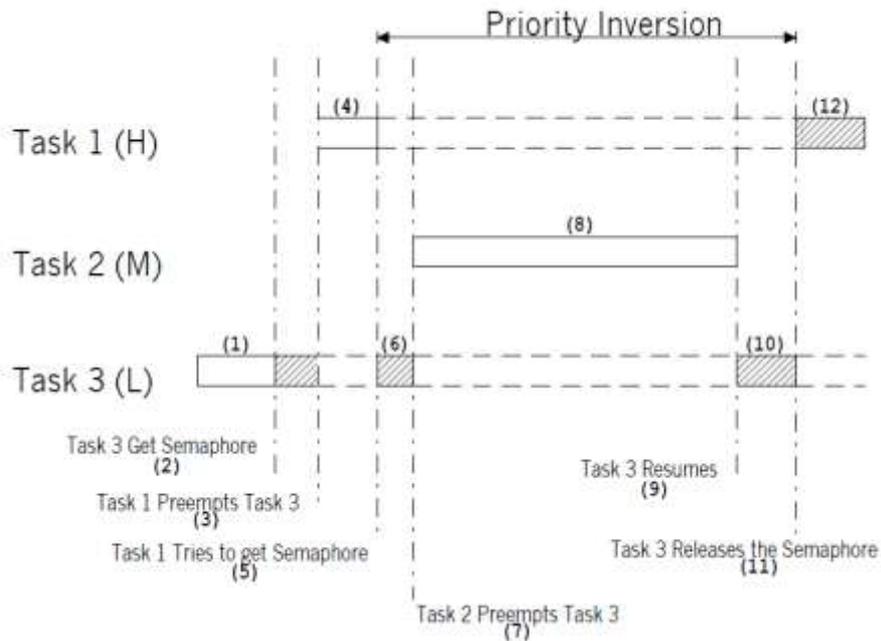



Figura 19. Inversão de prioridade. (LABROSSE, 2002).

Conforme a Figura 19, a tarefa 1 tem prioridade maior que a tarefa 2, que por sua vez tem prioridade maior que a tarefa 3. As tarefas 1 e 2 estão esperando por um evento e portanto a tarefa 3 está em execução (Figura 19-1). Em certo momento a tarefa 3 adquiri o semáforo que necessita para acessar um recurso compartilhado (Figura 19-2) com a tarefa 1, continuando sua operação sobre este recurso (Figura 19-3) até sofrer preempção da tarefa de maior prioridade (Figura 19-4). A tarefa 1 é, então, executada até o momento em que deseja acessar o recurso compartilhado por ela e pela tarefa 3 (Figura 19-5). Como a tarefa 3 possui o recurso, a tarefa 1 terá que esperar até que a tarefa 3 libere o semáforo. À medida que a tarefa 1 tenta adquirir o semáforo, o kernel verifica que o semáforo já está com "alguém", no caso a tarefa 3, dando continuidade a execução desta tarefa (Figura 19-6) e suspendendo a tarefa 1. Então, a tarefa 3 sofre preempção da tarefa 2 por que o evento que esta estava esperando ocorreu (Figura 19-7). A tarefa 2 opera sobre o evento (Figura 19-8) e quando termina sua execução, libera a CPU para a tarefa 3 (Figura 19-9). A

tarefa 3 executa seu trabalho, ainda possuindo o recurso (Figura 19-10) e então libera o semáforo para a execução da tarefa 1 (Figura 19-11). Neste ponto o kernel sabe que a tarefa de maior prioridade requer o semáforo e, portanto uma troca de contexto é feita para resumir a tarefa 1. A tarefa 1 tem o semáforo e pode acessar o recurso compartilhado (Figura 19-12).

Neste cenário a tarefa 1 reduziu, virtualmente, sua prioridade para a mesma prioridade da tarefa 3, porque teve que esperar um recurso que esta possuía. A situação se agravou ainda mais quando a tarefa 3 sofreu preempção da tarefa 2, atrasando ainda mais a execução da tarefa 1.

Conforme (LABROSSE, 2002), uma maneira simples de se resolver este problema é liberar o recurso compartilhado ao aumentar dinamicamente a prioridade da tarefa de menor prioridade, colocando-a como a tarefa de maior prioridade. Dessa maneira ela é executada e liberará o recurso para a tarefa original de maior prioridade e então sua prioridade é reduzida para retornar à sua prioridade original.

Além dessas operações, o kernel pode impedir preempções, usando um par de chamadas usadas para habilitar e desabilitar preempções no aplicativo. Esta característica pode ser útil se uma tarefa está executando uma região crítica de código[6], não podendo portanto sofrer preempção de outras tarefas.

5.7.1.3.Obtendo informações sobre as tarefas

[6] Uma região crítica de código é um código que deve ser tratado indivisivelmente, ou seja, uma vez que o código iniciou sua execução, ele não deve ser interrompido.

Os kernels fornecem rotinas que permitem os desenvolvedores acessar as informações das tarefas a partir de seus aplicativos. Estas informações são importantes para monitoramento e correção dos erros do aplicativo. A Tabela 4 ilustra algumas destas operações.

Tabela 4. Operações para se obter informações sobre as tarefas. (LI e YAO, 2003).

Operação	*Descrição da operação*
Get ID	Obtém a ID da tarefa requisitante.
Get TCB	Obtém o TCB da tarefa requisitante.

Obter a ID de uma tarefa em particular fornece mais informações do que obter seu TCB. Obter um TCB só fornece informações momentâneas de como está o contexto da tarefa e, como este é dinâmico, estas informações podem ser modificadas com o tempo.

5.7.2.Serviços de gerenciamento dos semáforos

As típicas operações com semáforos fornecidas pelos kernels para os desenvolvedores utilizarem em seus aplicativos incluem:

- Criar e apagar semáforos;
- Adquirir e liberar semáforos;
- Limpar a lista de tarefas-em-espera de um semáforo e;
- Obter informações sobre um semáforo.

5.7.2.1.Criando e apagando semáforos

As operações de criar e apagar semáforos são listadas na Tabela 5:

Tabela 5. Operações para criar e apagar semáforos. (LI e YAO, 2003).

Operação	Descrição da operação
Create	Cria um semáforo.
Delete	Apaga um semáforo.

De acordo com (LI e YAO, 2003), para se criar um semáforo deve-se especificar o estado inicial do semáforo (seu valor inicial) e a ordem de sua lista de tarefas-em-espera (FIFO ou por prioridade). A lista de tarefas-em-espera de um semáforo é inicialmente vazia na sua criação.

Semáforos podem ser apagados a partir de qualquer tarefa, especificando-se suas IDs e fazendo-se as chamadas para apagá-los. Quando um semáforo é apagado, as tarefas que estão bloqueadas em sua lista de tarefas-em-espera são desbloqueadas e mudam para o estado “pronto” ou “em execução” (se a tarefa desbloqueada tem a maior prioridade). Qualquer tarefa que tentar adquirir o semáforo apagado retorna um erro devido à inexistência do mesmo. Deve-se ter cuidado ao se apagar semáforos, pois se um semáforo for apagado quando estiver em uso, dados podem ser corrompidos, ou ainda problemas mais sérios podem surgir caso o semáforo esteja protegendo um recurso compartilhado ou uma seção crítica de código.

5.7.2.2.Adquirindo e liberando semáforos

A Tabela 6 lista as operações de adquirir e liberar semáforos.

Tabela 6. Operações para adquirir e liberar um semáforo. (LI e YAO, 2003).

Operação	***Descrição da operação***
Acquire	Adquire um (sinalizador ou chave) semáforo.
Release	Libera um (sinalizador ou chave) semáforo.

Segundo (LI e YAO, 2003), uma tarefa faz uma requisição para adquirir um semáforo de uma das seguintes maneiras:

- **Esperar para sempre** – a tarefa permanece bloqueada até o semáforo estar disponível para ser adquirido pela mesma.
- **Esperar por certo período** – a tarefa permanece bloqueada até adquirir o semáforo ou até expirar um intervalo de tempo pré-determinado pelo desenvolvedor. Neste ponto a tarefa é removida da lista de tarefas-em-espera do semáforo e colocada no estado “pronto” ou “em execução”.
- **Não esperar** – a tarefa requisita o semáforo, mas como este não está disponível, a tarefa não é bloqueada.

Um semáforo deve ser liberado com cuidado, pois pode resultar em perdas de acesso exclusivo a um recurso compartilhado ou um mau funcionamento nos dispositivos de I/O. Isto pode ocorrer, por exemplo, com uma tarefa que obteve acesso exclusivo a um recurso compartilhado ao adquirir um semáforo associado. Se outra tarefa, acidentalmente liberar este semáforo, uma terceira tarefa, que esteja

esperando o mesmo, pode obter acesso ao mesmo recurso, corrompendo-se desta forma os dados.

5.7.2.3.Limpando a lista de tarefas-em-espera

Alguns kernels suportam uma operação que limpa a lista de tarefas-em-espera de um semáforo, desbloqueando as tarefas que estejam na mesma. Esta operação é útil para transmissão de sinais a um grupo de tarefas e é denominada de operação *flush*. A Tabela 7 descreve a operação *flush*.

Tabela 7. Operação de limpeza da lista de tarefas-em-espera de um semáforo. (LI e YAO, 2003).

Operação	*Descrição da operação*
Flush	Desbloqueia todas as tarefas que aguardam um semáforo.

5.7.2.4.Obtendo a informação de um semáforo

Obter informações de um semáforo é importante para correção de erros e monitoramento do programa. A Tabela 8 identifica estas operações:

Tabela 8. Operações para adquirir informações sobre os semáforos. (LI e YAO, 2003).

Operação	*Descrição da operação*
Show info	Mostra informações gerais sobre semáforos.
Show blocked tasks	Mostra uma lista das IDs das tarefas que estão esperando por um semáforo.

5.7.3.Serviços de gerenciamento de fila de mensagens

As operações típicas executadas sobre as filas de mensagens são:

- Criar e apagar as filas de mensagens;
- Enviar e receber mensagens;
- Obter informação sobre as filas de mensagem.

5.7.3.1.Criando e apagando filas de mensagens

Filas de mensagens podem ser criadas e apagadas utilizando-se duas simples chamadas listadas na Tabela 9.

Tabela 9. Operações para criar e apagar uma fila de mensagens. (LI e YAO, 2003).

Operação	*Descrição da operação*
Create	Cria uma fila de mensagem.
Delete	Apaga uma fila de mensagem.

Conforme (LI e YAO, 2003), quando criada, uma fila de mensagens é tratada como um objeto global e não pertence a nenhuma tarefa em particular. O desenvolvedor da aplicação deve decidir o tamanho da fila, o tamanho máximo das mensagens e a ordem de espera das tarefas que estão bloqueadas na fila de mensagens.

Apagar uma fila de mensagens automaticamente desbloqueia todas as tarefas que estavam na lista de tarefas-em-espera da fila de mensagens; e as mensagens que estavam na lista são perdidas.

5.7.3.2.Enviando e recebendo mensagens

O uso mais comum de uma fila de mensagens é enviar e receber mensagens. As operações relacionadas ao envio e ao recebimento de mensagens estão listadas na Tabela 10.

Tabela 10. Operações de envio e recebimento de mensagens em uma fila de mensagens. (LI e YAO, 2003).

Operação	***Descrição da operação***
Send	Envia uma mensagem para a fila de mensagens.
Receive	Recebe uma mensagem de uma fila de mensagens.

Quando envia mensagens, o kernel geralmente preenche a fila de mensagens da cabeça para o rabo, na ordem FIFO, como mostra Figura 20. Cada mensagem é colocada no fim da fila.

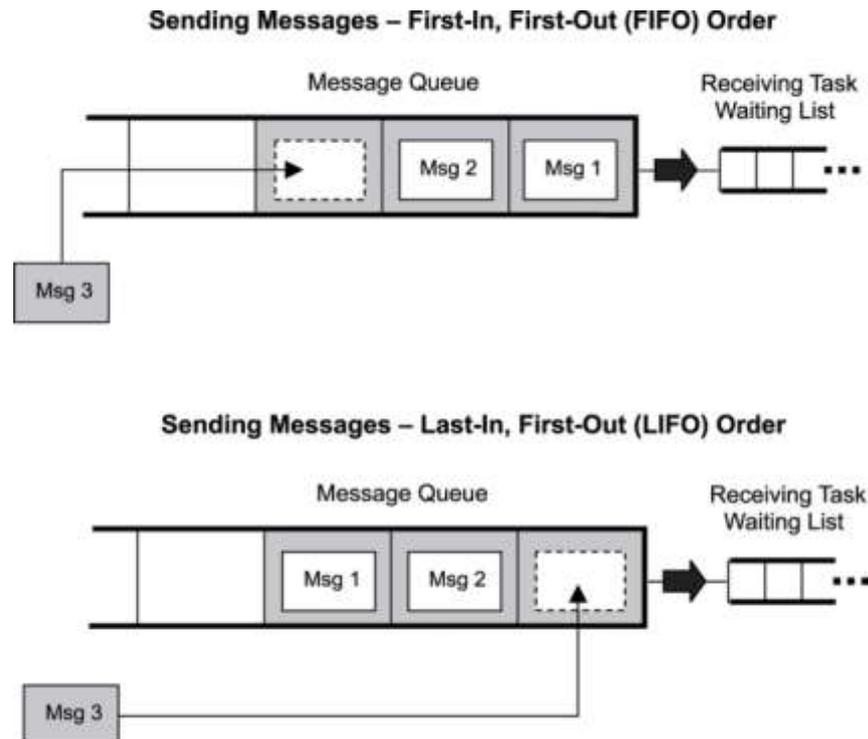



Figura 20. Envio de mensagens nas ordens FIFO e LIFO. (LI e YAO, 2003).

De acordo com (LI e YAO, 2003), alguns kernels permitem que mensagens urgentes vão direto para a cabeça da fila. Se todas as mensagens que cheguem à fila forem urgentes, todas irão para a cabeça da fila, o que caracterizaria uma ordenação do tipo LIFO (*last-in/first-out*).

Mensagens podem ser enviadas às filas de mensagem de três maneiras: não bloqueando as ISRs ou tarefas remetentes; bloqueando as tarefas remetentes por um intervalo de tempo; ou bloqueando as tarefas remetentes até que a mensagem seja recebida.

Mensagens que são enviadas a uma fila, sem causar bloqueio da tarefa ou ISR remetente, fazem com que a chamada de envio da mensagem retorne uma mensagem de erro para aquela tarefa ou ISR, se a fila de mensagens estiver cheia. Ao receber a mensagem de erro, a tarefa ou ISR continua sua execução.

Em alguns casos, entretanto, o sistema deve ser desenvolvido para que as tarefas sejam bloqueadas por um intervalo de tempo determinado, ou até receber a mensagem, quando uma mensagem é enviada para uma fila de mensagens cheia. Dessa forma, a tarefa bloqueada é colocada na lista de tarefas-em-espera da fila de mensagens para tarefas remetentes, na ordem FIFO ou baseada em sua prioridade, conforme mostra a Figura 21.

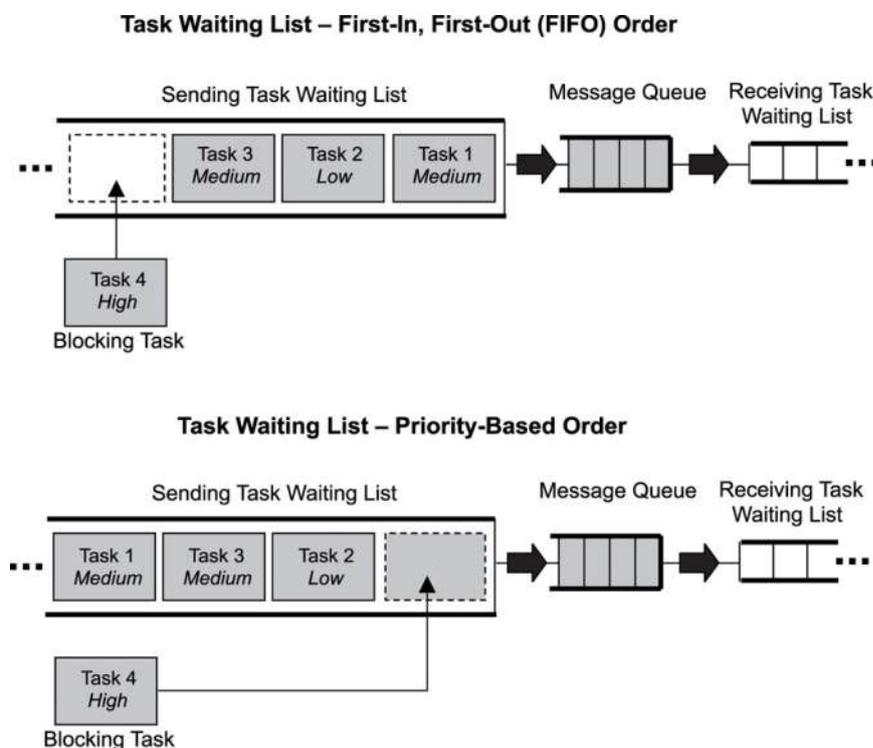



Figura 21. Lista de tarefas-em-espera da fila de mensagens na ordem FIFO e por prioridade. (LI e YAO, 2003).

Uma tarefa pode receber mensagens com políticas de bloqueio semelhantes ao envio de mensagens: não ser bloqueada, ser bloqueada por um intervalo de tempo, ou ser bloqueada até receber a mensagem. Neste caso, o bloqueio das tarefas destinatárias ocorre quando a fila de mensagens está vazia, sendo que as

tarefas destinatárias aguardam as mensagens na lista de tarefas-em-espera para tarefas destinatárias, ordenadas por prioridade ou FIFO.

Uma fila de mensagens se torna cheia se sua lista de tarefas-em-espera para tarefas destinatárias estiver vazia ou a taxa com que as mensagens chegam à fila de mensagens é maior do que a taxa com que as mensagens são removidas da fila. Somente quando a fila de mensagens está cheia, inicia-se o preenchimento da lista de tarefas-em-espera para tarefas remetentes. Por outro lado, o preenchimento da lista das tarefas-em-espera para tarefas destinatárias só é iniciado quando a fila de mensagens está vazia.

Mensagens podem ser lidas das filas de mensagens de duas maneiras: leitura destrutiva e leitura não destrutiva. Na leitura destrutiva, quando uma tarefa recebe com sucesso uma mensagem da fila, a tarefa remove permanentemente a mensagem do buffer de armazenamento da fila de mensagens. Numa leitura não destrutiva, a tarefa destinatária “copia” a mensagem da fila de mensagens sem removê-la. Poucas implementações de kernel possuem leitura não-destrutiva.

5.7.3.3.Obtendo informações sobre as filas de mensagens

Como as outras operações de obtenção de informações de um objeto do kernel em particular, estes tipos de chamadas servem para correção dos erros do programa e monitoramento do mesmo. A

Tabela **11** lista as operações típicas para este fim.

Tabela 11. Operações para obtenção de informações de uma fila de mensagens. (LI e YAO, 2003).

Operação	*Descrição da operação*
Show queue info	Obtém informações de uma lista de mensagens.
Show queue´s task-waiting list	Obtém a lista de tarefas que estão na lista de tarefas-em-espera de uma fila de mensagens.

5.8.A base de tempo de um RTOS

Nas aplicações de sistemas embarcados, as tarefas do sistema e do usuário geralmente agendam suas atividades para serem executadas no decorrer do tempo. O agendamento futuro destas atividades é feito através de temporizadores, presentes no hardware dos sistemas embarcados, e utilizando chamadas de temporização do kernel.

5.8.1.Os Temporizadores

De acordo com (LI e YAO, 2003), os temporizadores, também conhecidos como temporizadores de intervalo programável (PIT), são dispositivos utilizados para a contagem de eventos, para a indicação de tempo decorrido, para geração de eventos periódicos a taxas controláveis e outras aplicações que resolvam sistemas com problemas de controle do tempo.

Apesar da diferença entre os diversos chips temporizadores existentes, algumas características são comuns entre todos: um sinal de *clock* de entrada com freqüência fixa, registradores de controle programáveis e um sinal de saída.

A temporização de um evento é determinada pela taxa de interrupção do temporizador, que é o número de interrupções geradas por segundo. A taxa de interrupção do temporizador é calculada como função da freqüência do *clock* de entrada e é configurada em um dos registradores de controle do temporizador. Neste registrador contador é carregado o valor que determinará quando será a próxima interrupção do temporizador. Este valor é incrementado ou decrementado (dependendo do funcionamento do registrador contador) a cada ciclo de *clock*. Quando este valor ultrapassa seu valor máximo (ou mínimo, no caso de um registrador contador "decrementador"), ocorre o que se chama de estouro (*overflow*) do registrador contador e um sinal de saída é ativado, disparando uma interrupção ou, em certas aplicações, apenas avisando o processador de que houve um estouro na contagem do registrador contador.

Se a interrupção do temporizador for periódica consegue-se estabelecer uma medida para o tempo decorrido, permitindo ao sistema trabalhar com certa exatidão temporal. Para que a interrupção seja periódica é necessário reiniciar o temporizador, via software, carregando novamente, com o mesmo valor inicial, o registrador contador do temporizador.

O chip temporizador deve ser inicializado, via software, durante a inicialização do sistema. Toda informação referente à sua configuração encontra-se no manual do usuário do fabricante.

5.8.2.O tick do relógio do sistema

Conforme (LABROSSE, 2002), cada interrupção gerada pelo temporizador é denominada de *clock tick*, ou simplesmente *tick*, e representa uma unidade de tempo. O tempo entre duas interrupções é especificado no aplicativo e geralmente está entre 10 e 200 ms. Quanto mais rápido for a taxa de *tick*, mais alto será o *overhead* imposto ao sistema.

A interrupção do *tick* permite ao kernel atrasar a execução de suas tarefas, por um número inteiro de *ticks,* e indicar o término do tempo de espera por um evento quando uma tarefa está esperando pela ocorrência do mesmo. Isto é possível porque a interrupção do *tick* conduzirá a CPU para a execução de sua rotina (ISR), onde estará codificada uma função que invocará os algoritmos do kernel para avaliar a necessidade de se realizar o escalonamento de suas tarefas. Esta capacidade de gerar eventos periódicos torna o PIT o coração de muitos kernels de tempo real.

A resolução do atraso de uma tarefa é de um *tick*. No entanto, isto não significa que a precisão será de exatamente um *tick*. Esta precisão está sujeita a variação (denominada de jitter) do tempo de execução da ISR e das tarefas de maior prioridade ao causarem preempção de uma tarefa periódica de menor prioridade. As figuras Figura 22, Figura 23 e Figura 24 exemplificam essa imprecisão (jitter) para uma tarefa que se atrasa de um *tick*. As áreas sombreadas indicam o tempo de execução de cada operação. É importante notar que o tempo de execução varia para cada tipo de operação: loops, condições, dentre outras. O tempo de processamento da ISR após o *tick* foi exagerado para mostrar que ele também está sujeito à variação no seu tempo de execução.

No primeiro caso, a Figura 22 mostra uma situação onde tarefas de maior prioridade e ISRs são executadas antes de uma tarefa, cujo atraso é de um *tick.* Neste caso, a tarefa tenta se atrasar de 20 ms, mas por causa da sua prioridade, é executada em intervalos variáveis.

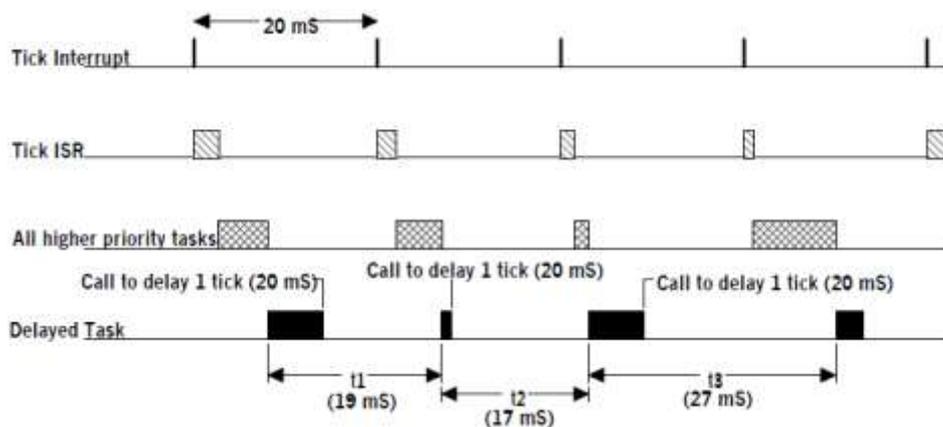



Figura 22. Atrasando uma tarefa por um tick - Primeiro caso. (LABROSSE, 2002).

O segundo caso (Figura 23) mostra uma situação onde o tempo de execução de todas as tarefas de maior prioridade e ISRs são um pouco menores do que um *tick* do relógio do sistema. A tarefa que se atrasou por um *tick* é executada quase que imediatamente. Por este motivo se houver a necessidade de se atrasar uma tarefa por um número mínimo de *ticks*, deve-se sempre adicionar um *tick* extra. Em outras palavras, se houver a necessidade de se atrasar uma tarefa por no mínimo 5 *ticks*, deve ser especificado 6 *ticks* de atraso.

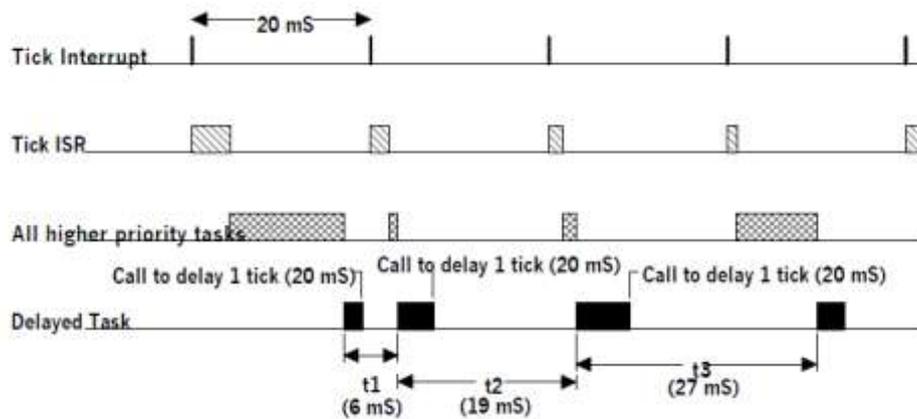



Figura 23. Atrasando uma tarefa por um tick - Segundo caso. (LABROSSE, 2002).

A Figura 24 mostra uma situação em que o tempo de execução de todas as tarefas de maior prioridade e das ISRs ultrapassam um *tick* do sistema. Neste terceiro caso, a tarefa tenta se atrasar por um *tick*, mas só consegue ser executada dois *tick* mais tarde, ultrapassando seu limite de tempo. Ultrapassar o limite de tempo pode ser aceitável em algumas aplicações, mas na maioria dos casos não é.

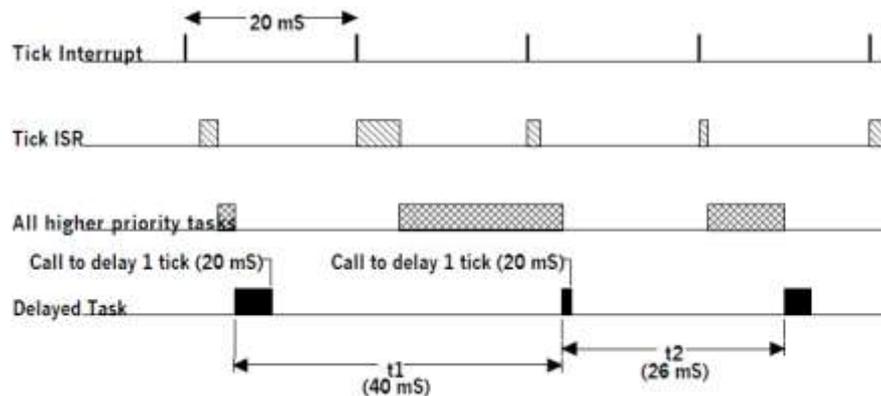



Figura 24. Atrasando uma tarefa por um tick - Terceiro caso. (LABROSSE, 2002).

Algumas soluções podem amenizar a ocorrência destes problemas, dentre elas estão:

- Aumentar a taxa de *clock* do microprocessador;

- Aumentar o tempo entre duas interrupções do *tick*;
- Rearranjar as prioridades das tarefas;
- Evitar usar floating-points;
- Utilizar um compilador que possua uma melhor otimização do código;
- Escrever os códigos críticos em tempos em assembly;
- Migrar para um processador mais rápido da mesma família.

Mas apesar das melhorias que possam ser alcançadas, esta variação na resolução do atraso das tarefas sempre ocorrerá.

5.8.3.As rotinas de interrupção para RTOSs

Conforme dito na seção 5.3 quando ocorre uma interrupção, a CPU passa a executar uma rotina que está codificada num endereço específico da memória de programa. Este endereço está intimamente relacionado com a interrupção reconhecida (qual sinal de interrupção foi ativado) e é especificado pelo fabricante. A rotina localizada neste endereço é a rotina de interrupção (ISR) e possui prioridade maior do que qualquer tarefa do sistema.

Na inicialização do PIT, feita durante a inicialização do sistema, deve-se instalar a rotina de interrupção (ISR) que será invocada para tratar do tempo do RTOS quando ocorrer uma interrupção do temporizador. Esta rotina deve executar, dentre outras atividades, as seguintes ações:

- **Atualizar o relógio do sistema** – o tempo decorrido deve ser medido em *ticks* e indicar quanto tempo o sistema está em execução desde sua inicialização;

- **Fazer a chamada da função do kernel que notificará a passagem do tempo para todas as tarefas do RTOS em cada *tick*** – esta função deverá trabalhar em conjunto com os TCBs das tarefas, mantendo-os sempre atualizados sobre quanto tempo uma tarefa ainda permanecerá bloqueada;
- **Fazer a chamada da função do kernel que invocará o escalonador para a execução da tarefa de maior prioridade que estiver no estado "pronto"** – depois da atualização dos TCBs, o escalonador deve ser invocado para verificar se houveram alterações nos estados das tarefas no decorrer do tempo e, caso seja necessário, executar a tarefa de maior prioridade que estiver pronta para a execução.
- **Reconhecer a interrupção, reiniciar o temporizador e retornar da interrupção.**

A Figura 25 ilustra bem o processo de medição do tempo de um RTOS. O temporizador vai sendo incrementado a cada período de um sinal de clock. Quando este incremento ultrapassa seu valor máximo, um sinal (interrupção) é enviado para a CPU. Então, o processador passa a executar a rotina específica (ISR) para aquele sinal de interrupção. Nesta rotina, todas as tarefas serão avisadas da passagem do tempo; o escalonador será invocado para avaliar o estado das tarefas e executar a tarefa "pronta" de mais alta prioridade; e o temporizador será reiniciado com o valor de tempo que se quer ter como resolução do *tick*. Dessa forma o sistema torna-se periódico sendo possível medi-lo com certa precisão.

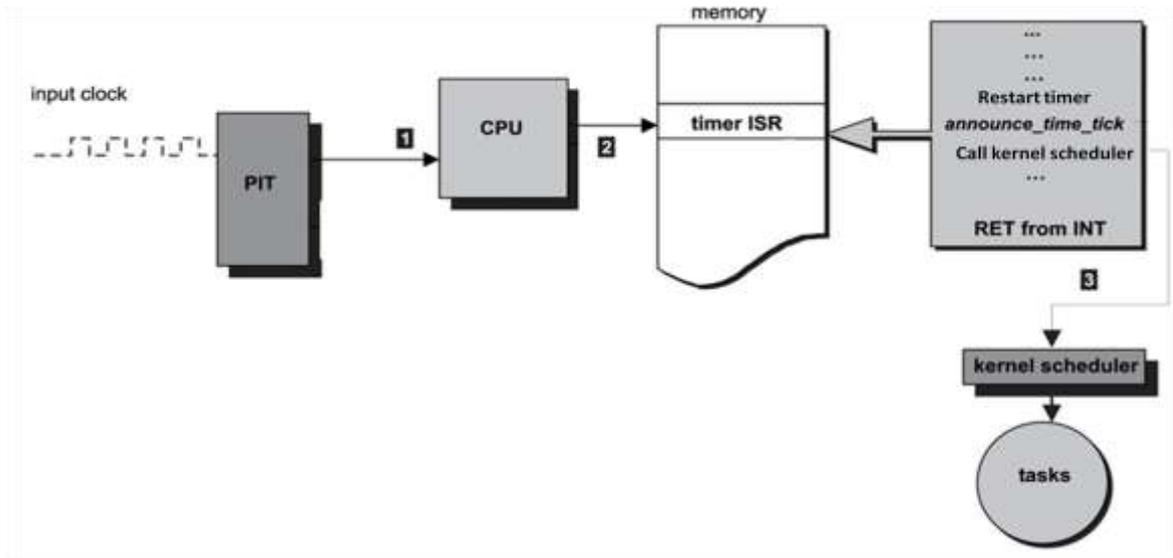



Figura 25. A base de tempo de um RTOS.

Devido à imprecisão dos *ticks* vistos pelas tarefas, apresentada na seção 5.8.2, a ISR responsável pela base de tempo do RTOS deve ser pequena e conduzir o mínimo de atividades possíveis para amenizar ao máximo sua imprecisão.

6.O µC/OS-II

Existem mais de 150 sistemas operacionais de tempo real no mercado atualmente. Alguns são grátis, outros variam de Us$70,00 a Us$30.000,00. Além disso, os vendedores de RTOSs podem ainda cobrar royalties que variam de Us$5,00 até Us$70,00 por unidade de chip produzido comercialmente.

Apesar de ser pago, o µC/OS-II foi escolhido para a realização deste trabalho pelos seguintes motivos:

- Baixo custo para uso comercial, livre de royalties;
- Não possui custo nenhum em uso educacional;
- Fornecimento de todo o código fonte;
- Código escrito em ANSI C, limpo e padronizado, o que facilita a adaptação deste sistema operacional para diversos tipos de processadores;
- Facilidade de acesso ao código fonte, podendo ser feito o download na página do fabricante Micrium;
- Extensa documentação, possuindo livro didático cujo nome se encontra em (LABROSSE, 2002);
- Existência de suporte técnico;
- Atende aos requisitos de sistemas críticos em segurança;
- Reconhecido por grandes empresas como NXP, IAR Systems, ARM, dentre outras;
- Utilizado em centenas de produtos em todo o mundo, tornando-se um dos mais populares da atualidade.

Toda a estrutura do kernel do µC/OS-II, bem como o uso de seus serviços e objetos, encontra-se descrita com detalhes em (LABROSSE, 2002).

6.1.Características do µC/OS-II

- PREEMPTIVO

  O µC/OS-II é um kernel de tempo real totalmente preemptivo-prioritário, sempre executando a tarefa de maior prioridade que está pronta para ser executada.

- MULTI-TASKING

  O µC/OS-II até a versão 2.80 gerenciava até 64 tarefas, sendo 8 tarefas reservadas para o sistema e 56 para o usuário. Na versão 2.86, ele pode gerenciar até 255 tarefas.

- DETERMINÍSTICO

  O tempo de execução de todas as funções e serviços do µC/OS-II é determinístico, ou seja, o desenvolvedor sabe quanto tempo o kernel vai demorar para executar uma função ou serviço.

- SERVIÇOS

  O µC/OS-II fornece uma variedade de objetos e serviços como caixas de correio, filas de mensagens, semáforos, particionamento da memória, funções relacionadas ao tempo, dentre outros.

- GERENCIAMENTO DE INTERRUPÇÕES

  As interrupções podem suspender a execução de uma tarefa e, se uma tarefa de maior prioridade fica pronta para execução a partir de uma interrupção, ela

é executada assim que a interrupção (ou todas as interrupções que estejam aninhadas) termine sua execução. Até 255 interrupções podem ser aninhadas no µC/OS-II.

6.2.Estados das tarefas e os serviços do µC/OS-II

Os estados das tarefas do µC/OS-II seguem a máquina de estado representada na Figura 26.

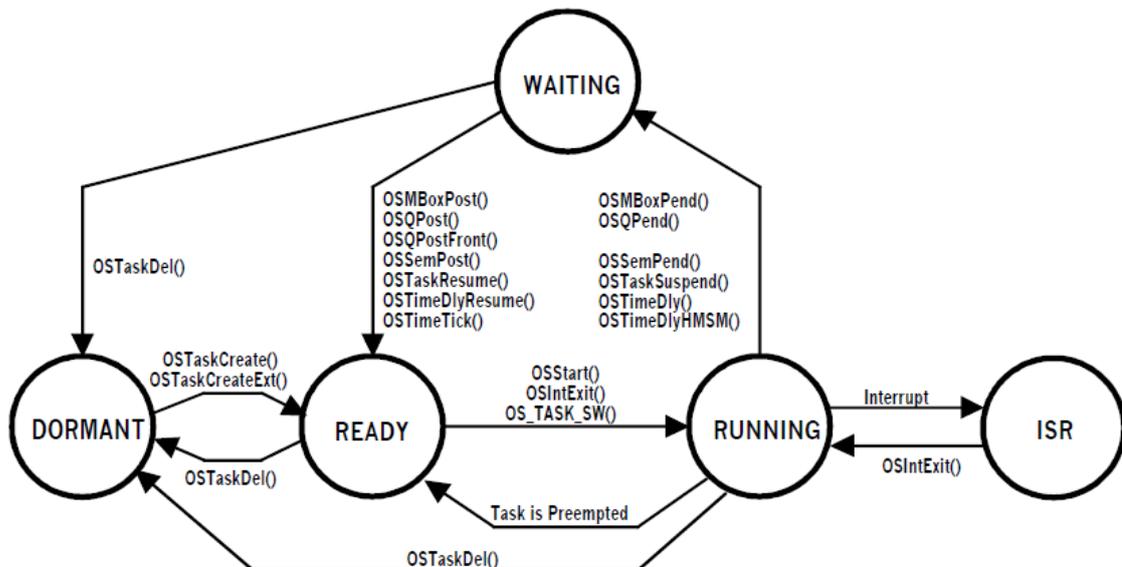



Figura 26. Estados das tarefas no µC/OS-II. (LABROSSE, 2002).

Segundo (LABROSSE, 2002), uma tarefa no estado *dormant* corresponde a uma tarefa que se encontra na memória de programa (RAM ou ROM), mas não está disponível para o µC/OS-II. Uma tarefa se torna disponível para o µC/OS-II através das chamadas dos serviços do kernel OSTaskCreate() ou OSTaskCreateExt(). Estes serviços são simplesmente usados para criar uma tarefa, ou seja, dizer ao µC/OS-II o endereço inicial de suas tarefas, a prioridade a ser dada para a tarefa a ser criada,

quanto de espaço de memória terá sua pilha e assim por diante. Quando uma tarefa é criada, ela torna-se pronta para a execução e é colocada no estado pronto (*ready state*).

No µC/OS-II, as tarefas podem ser criadas antes do início do tratamento das multi-tarefas ou dinamicamente por uma tarefa em execução. Se uma tarefa criada a partir de outra tarefa em execução tiver prioridade maior do que a tarefa que a acabou de criá-la, aquela toma imediatamente o controle da CPU. Uma tarefa pode colocar a si mesmo, ou outra tarefa, novamente no estado *dormant* através do serviço OSTaskDel().

O tratamento multi-tarefa é iniciado chamando-se o serviço OSStart(). OSStart() deve ser chamado apenas uma vez, durante a inicialização do RTOS e inicia a execução da tarefa de maior prioridade que foi criada durante a inicialização do código. Esta tarefa de maior prioridade é então colocada no estado "em execução" (*running state*).

Como visto nas seções anteriores, apenas uma tarefa pode estar em execução na CPU. Uma tarefa que está pronta para a execução, mas não possui a maior prioridade é colocada no estado "em espera" (*waiting state*), que corresponde ao estado bloqueado apresentado na seção 5.6.1, aguardando sua execução quando o kernel lhe der a oportunidade.

A tarefa em execução pode atrasar ela mesma, por certo período tempo, chamando os serviços OSTimeDly() ou OSTimeDlyHMSM(). Esta tarefa seria colocada no estado "em espera" (*waiting state)* até que o tempo especificado expirasse. Estes serviços forçam uma troca de contexto imediata para a próxima tarefa de maior prioridade, que esteja pronta para a execução. A tarefa atrasada

torna-se pronta para execução, no momento em que o tempo especificado para o atraso expira, através da chamada do kernel OSTimeTick(). OSTimeTick() é uma função interna do kernel, não sendo chamada pelo desenvolvedor da aplicação.

Uma tarefa em execução pode, também, ser colocada em espera até que um evento ocorra chamando os serviços OSSemPend(), OSMboxPend(), OSQPend(), OSFlagPend() ou OSMutexPend(). Cada tipo de evento (semáforo, caixa de correio, fila de mensagens, dentre outros) está relacionado com um serviço específico. Se o evento ainda não ocorreu, a tarefa que invocou algum destes serviços é colocada no estado "em espera" até a ocorrência do evento.

Quando uma tarefa passa a esperar um evento, a próxima tarefa pronta de mais alta prioridade toma imediatamente o controle da CPU. Uma tarefa que espera um evento torna-se pronta para a execução novamente quando o evento ocorre ou quando o seu tempo de espera expira. A ocorrência de um evento pode ser sinalizada por uma tarefa ou uma ISR chamando algum dos seguintes serviços: OSQPost(), OSSemPost(), OSMboxPost(), OSMboxPostOpt(), OSMutexPost(), OSFlagPost(), OSQPostOpt(), OSQPostFront().

Uma tarefa em execução sempre pode ser interrompida, a menos que alguma tarefa ou o µC/OS-II tenha desabilitado as interrupções do hardware. A tarefa entra então no estado "execução da ISR" (*ISR running state*). Quando uma interrupção ocorre, a execução da tarefa é suspensa e a ISR obtém o controle da CPU. Uma ISR pode fazer uma ou mais tarefas mudarem seus estados para estados "prontos" através da sinalização de eventos. Por isso, antes de retornar de uma interrupção, o µC/OS-II verifica se a tarefa interrompida é ainda a tarefa de maior prioridade. Se a ISR fizer uma tarefa de maior prioridade pronta para execução, esta é executada

imediatamente após o retorno da interrupção, senão a tarefa interrompida obtém novamente o controle da CPU.

6.3.A estrutura de arquivos do μC/OS-II

A estrutura de arquivos do μC/OS-II e sua relação com o hardware estão representadas na Figura 27.

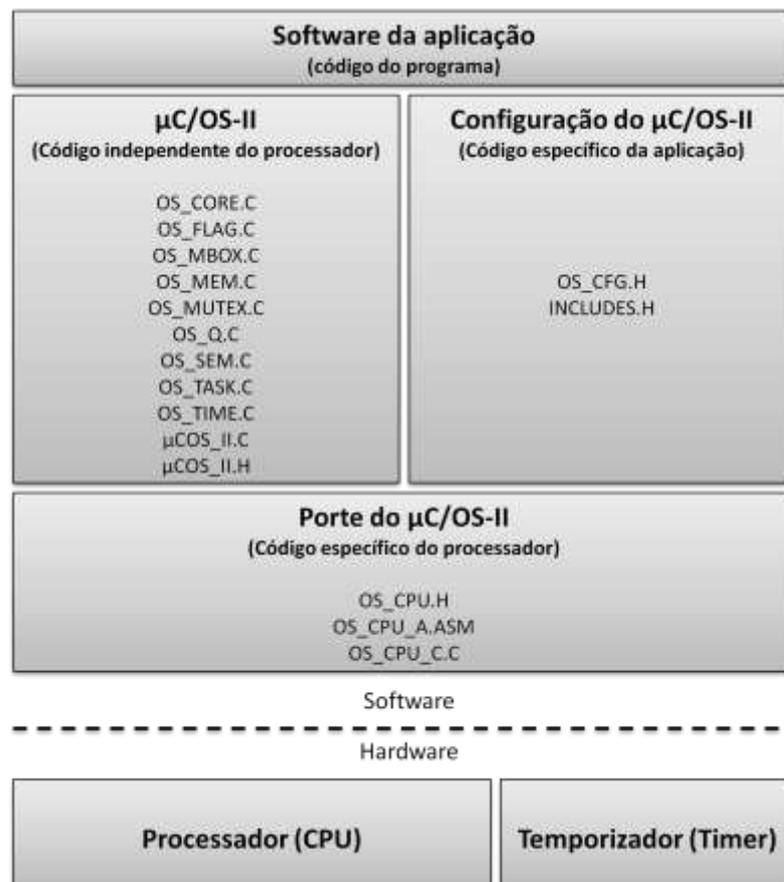



Figura 27. Estrutura de arquivos do μC/OS-II. (LABROSSE, 2002).

Nos arquivos independentes do processador no qual o μC/OS-II será adaptado, estão os códigos dos serviços, dos objetos e das chamadas do kernel que permitem a ele gerenciar as tarefas da aplicação. Nos arquivos de configuração do μC/OS-II,

que são específicos para cada aplicação, o kernel permite ao programador selecionar e quantificar apenas os serviços e objetos que ele deseja utilizar. Isto traz uma grande vantagem, na medida em que se evita o desperdício de memória do hardware, que é um recurso bem escasso em sistemas embarcados.

Os arquivos específicos do processador são aqueles codificados para manipular seus registradores, por exemplo, numa troca de contexto entre tarefas, e por isso diferem para cada hardware utilizado. Estes são os principais arquivos a serem modificados para adaptar o μC/OS-II em um microprocessador. A adaptação de um kernel de tempo real em um microprocessador é denominada de porte. A portabilidade do μC/OS-II é, segundo (LABROSSE, 2002) relativamente fácil perante outros RTOSs, pois ele foi feito para ser portável sendo que a maioria de seu código é escrito em ANSI C, a linguagem C padrão. No entanto, existe a necessidade de se escrever alguns códigos específicos do processador em linguagem de montagem (Assembly), e para isto é necessário se ter bom conhecimento do hardware alvo e de suas instruções.

6.4.A portabilidade do μC/OS-II para o PIC18F

Nathan Brown fez a adaptação, ou portabilidade, do μC/OS-II para o PIC18F em 30 de outubro de 2002. Em 20 de dezembro de 2002 lançou uma nova versão onde havia implementado algumas melhorias. A versão do μC/OS-II que foi portada naquela época é a 2.51. Este porte feito por Nathan Brown pode ser encontrado em (BROWN, 2005), que também possui um link no site da Micrium, fabricante do μC/OS-II.

Muitas versões do µC/OS-II saíram desde 2002 até hoje, e sua portabilidade para o PIC18F nunca foi atualizado no site do fabricante Micrium, nem no site de Nathan Brown. Hoje o µC/OS-II está na versão 2.86 com muitas correções e adição de novos serviços e objetos. Como é preferível se trabalhar com a versão do µC/OS-II mais livre de bugs, foi feita uma atualização do porte deste sistema operacional para se utilizar a versão mais nova, versão 2.86, que está descrita no APÊNDICE C deste trabalho.

A portabilidade feita por Nathan Brown não será descrita aqui, sua explicação encontra-se no seu site pessoal em (BROWN, 2005). Em (LABROSSE, 2002) têm-se um capítulo inteiro explicando sobre a portabilidade do µC/OS-II, com alguns pseudocódigos que auxiliam no desenvolvimento de um porte. Este trabalho traz no seu Apêndice C apenas as etapas necessárias para fazer o µC/OS-II rodar em um PIC18F, com as alterações necessárias para que a versão utilizada seja a 2.86.

7.MATERIAIS E MÉTODOS

A análise do uso de um sistema operacional de tempo real em um software será feita através de uma comparação entre um software que utilize um sistema operacional de tempo real (RTOS) e outro que não utilize o RTOS. Para isto, foram desenvolvidos dois softwares diferentes, em linguagem de programação C, para a seguinte aplicação: contador crescente/decrescente de 4 dígitos, que incremente/decremente a contagem a cada 10 Hz ou 100 ms. O hardware a ser utilizado nesta aplicação será a placa de desenvolvimento McLab2, presente no laboratório de mecatrônica da Faculdade de Engenharia Mecânica da Unicamp.

Neste capítulo serão descrito: os softwares utilizados para o desenvolvimento dos programas; a placa de desenvolvimento McLab2; a descrição detalhada do contador; a metodologia utilizada para se fazer a comparação entre os programas; o programa do contador crescente/decrescente sem o sistema operacional de tempo real; e o programa do contador crescente/decrescente com o sistema operacional de tempo real.

7.1.Softwares utilizados

Os softwares utilizados para o desenvolvimento dos programas com, e sem, o sistema operacional foram:

- **Ambiente de desenvolvimento:** MPLAB® IDE, versão 8.20a, cujo fabricante é a Microchip.
- **Compilador:** MPLAB® C18 C COMPILER, versão 3.30 acadêmica, cujo fabricante também é a Microchip.

- **Sistema Operacional de Tempo Real:** µC/OS-II, versão 2.86, cujo fabricante é a Micrium.

7.2.Placa de desenvolvimento McLab2

A placa de desenvolvimento McLab2 foi fabricada pela Mosaico Engenharia. Os recursos que a placa oferece são:

- LCD alfanumérico;
- Displays de leds de 7 segmentos;
- Teclas e leds;
- Buzzer;
- Memória serial EEPROM 24C04 (protocolo I^2C);
- Comunicação serial RS232;
- Conversão A/D;
- Sensor de temperatura;
- Aquecedor;
- Ventilador;
- Tacômetro;
- Leitura de jumpers;
- Conector de expansão contendo 15 I/O's;
- Botão de reset manual;
- Gravação in-circuit compatível com McFlash.

Para o desenvolvimento do contador crescente/decrescente de 4 dígitos, com 100 ms de resolução, foram utilizados somente as teclas (teclado) e os displays de 7

segmentos. O microcontrolador utilizado na placa como objeto central foi o PIC18F452.

Apenas os componentes da placa, bem como os do PIC18F452, que foram utilizados no desenvolvimento do contador, serão descritos nesta seção. Informações adicionais podem ser encontradas no manual de utilização da placa MClab2 e no *datasheet* do PIC18F452. A Figura 28 ilustra a placa McLab2.

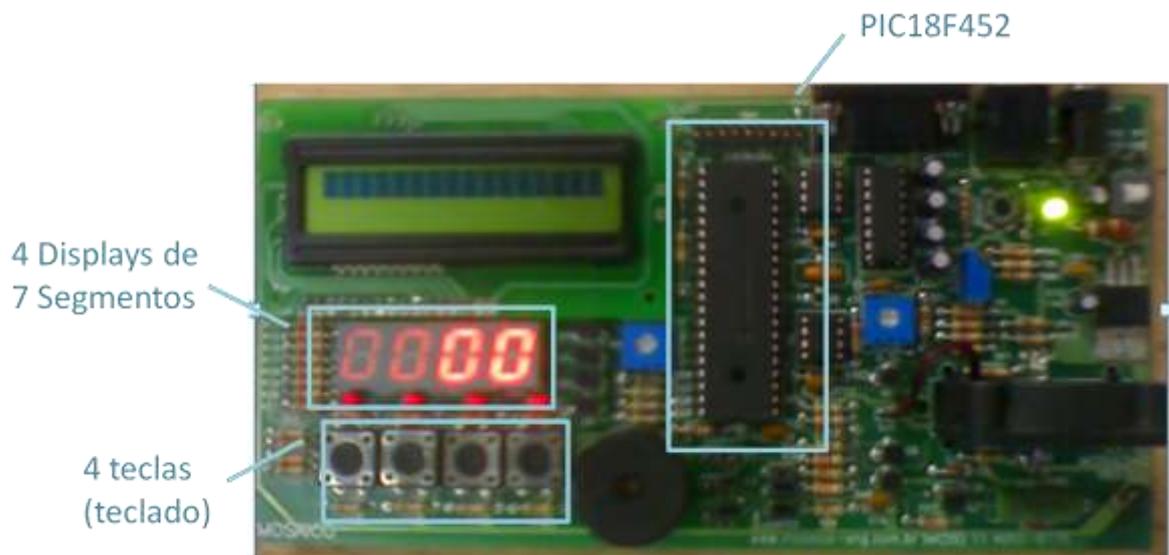



Figura 28. Placa de desenvolvimento McLab2.

7.2.1.PIC18F452

As principais característica do microcontrolador PIC18F452 são:

- 8 bits de barramento de dados;
- 32K de memória de programa tipo Flash;
- 1.5K bytes de memória de dados volátil (RAM);
- 256 bytes de memória de dados não volátil (E^2PROM);
- 33 I/O´s;

- 4 timers (3 de 16 bits, 1 de 8 bits);
- 2 Capture/Compare/PWM de 10 bits;
- 2 periféricos de comunicação digitais: 1 A/E/USART e 1 MSSP (PSI/ I²C);
- PSP;
- ICD;
- 8 canais de conversão A/D com 10 bits cada;
- 10 MIPS de velocidade do Processador;
- 40MHz de velocidade máxima.

A pinagem deste microcontrolador está representada na Figura 29.

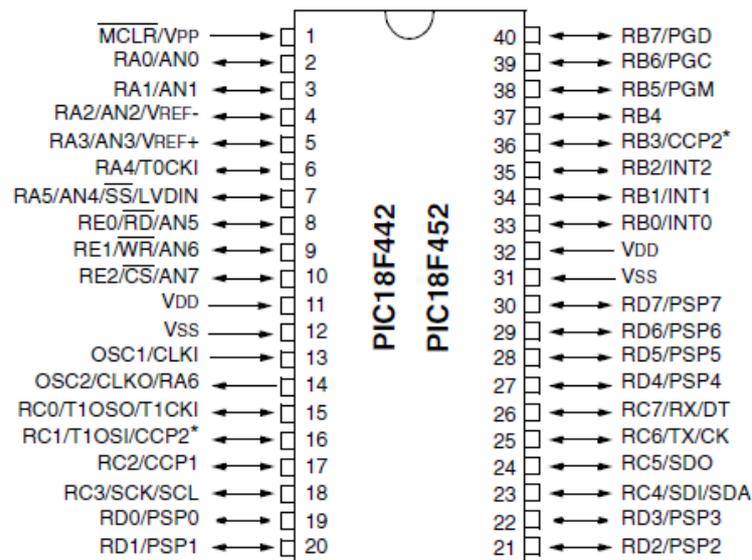



Figura 29. Pinagem do PIC18F452. (DATA, 2006).

Para o desenvolvimento do contador foram utilizados os seguintes periféricos do PIC18F452:

- 8 I/O´s – PORTB (RB0 a RB7) e PORTD (RD0 a RD7);
- 1 Timer – TIMER0.

7.2.1.1.PORTB

Segundo (DATA, 2006), o *datasheet* do PIC18F452, o PORTB é uma porta de 8 vias digitais bidirecionais (para a entrada/saída de dados), sendo portanto de 8 bits. Além disso, estes pinos são compartilhados, podendo ser utilizados como pinos para geração de interrupções e para programação dos dispositivos seriais, caso sejam configurados para estes fins. A Tabela 12 mostra a função de cada um dos bits do PORTB.

Tabela 12. Configuração dos pinos do PORTB do PIC18F452. (DATA, 2006)

Name	Bit#	Buffer	Function
RB0/INT0	bit0	TTL/ST(1)	Input/output pin or external interrupt input0. Internal software programmable weak pull-up.
RB1/INT1	bit1	TTL/ST(1)	Input/output pin or external interrupt input1. Internal software programmable weak pull-up.
RB2/INT2	bit2	TTL/ST(1)	Input/output pin or external interrupt input2. Internal software programmable weak pull-up.
RB3/CCP2(3)	bit3	TTL/ST(4)	Input/output pin or Capture2 input/Compare2 output/PWM output when CCP2MX configuration bit is enabled. Internal software programmable weak pull-up.
RB4	bit4	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB5/PGM(5)	bit5	TTL/ST(2)	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. Low voltage ICSP enable pin.
RB6/PGC	bit6	TTL/ST(2)	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. Serial programming clock.
RB7/PGD	bit7	TTL/ST(2)	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. Serial programming data.

Legend: TTL = TTL input, ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: A device configuration bit selects which I/O pin the CCP2 pin is multiplexed on.
4: This buffer is a Schmitt Trigger input when configured as the CCP2 input.
5: Low Voltage ICSP Programming (LVP) is enabled by default, which disables the RB5 I/O function. LVP must be disabled to enable RB5 as an I/O pin and allow maximum compatibility to the other 28-pin and 40-pin mid-range devices.

Para o desenvolvimento do contador crescente/decrescente, o PORTB foi configurado para trabalhar como uma via bidirecional (entrada/saída de dados). Dessa forma o PORTB deve ser configurado para ler um valor de entrada ou escrever um valor de saída, dependendo da configuração de sua direção.

O PORTB é controlado por três registradores:

- PORTB – registrados dos dados da porta;
- TRISB – registrador da direção da porta;
- LATB – registrador que armazena informação do valor atual de cada bit da porta para que este não seja afetada por nenhum dispositivo externo conectado a um pino.

Para se configurar um bit do PORTB como entrada digital, deve-se atribuir o valor 1 ao bit correspondente do TRISB. Para que um bit do PORTB seja saída digital, deve-se atribuir o valor 0 ao bit correspondente de TRISB.

7.2.1.2.PORTD

Conforme (DATA, 2006), a PORTD é também uma via bidirecional de 8 bits, tendo seu próprio registrador de dados (PORTD), de direção dos dados (TRISD) e seu registrador armazenador (LATD). Operando como uma via bidirecional, a PORTD é similar à PORTB. Para esta aplicação, o tratamento dado a PORTD foi o mesmo que o dado à PORTB, ou seja, uma via bidirecional de 8 bits. A Tabela 13 mostra a função de cada um dos bits da PORTD:

Tabela 13. Função dos bits da PORTD do PIC18F452. (DATA, 2006).

Name	Bit#	Buffer Type	Function
RD0/PSP0	bit0	ST/TTL(1)	Input/output port pin or parallel slave port bit0.
RD1/PSP1	bit1	ST/TTL(1)	Input/output port pin or parallel slave port bit1.
RD2/PSP2	bit2	ST/TTL(1)	Input/output port pin or parallel slave port bit2.
RD3/PSP3	bit3	ST/TTL(1)	Input/output port pin or parallel slave port bit3.
RD4/PSP4	bit4	ST/TTL(1)	Input/output port pin or parallel slave port bit4.
RD5/PSP5	bit5	ST/TTL(1)	Input/output port pin or parallel slave port bit5.
RD6/PSP6	bit6	ST/TTL(1)	Input/output port pin or parallel slave port bit6.
RD7/PSP7	bit7	ST/TTL(1)	Input/output port pin or parallel slave port bit7.

Legend: ST = Schmitt Trigger input, TTL = TTL input

Note 1: Input buffers are Schmitt Triggers when in I/O mode and TTL buffer when in Parallel Slave Port mode.

7.2.1.3.TIMER0

TIMER0 é, de acordo com (DATA, 2006), um contador/temporizador programável que pode operar com 8 ou 16 bits, incrementado por um sinal de *clock* derivado do *clock* do processador ($F_{clkprocessdor}$/4), ou de um *clock* externo. Esse sinal de *clock* passa por uma pré-escala programável de 8 bits antes de incrementar o valor do TIMER0.

O registrador que controla a configuração deste contador é o T0CON (TIMER0 CONTROL REGISTER) e as funções de cada bit deste registrador é apresentado na Figura 30:

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0

bit 6 **T08BIT**: Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

bit 5 **T0CS**: Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)

bit 4 **T0SE**: Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Timer0 Prescaler Assignment bit
1 = TImer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.

bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits
111 = 1:256 prescale value
110 = 1:128 prescale value
101 = 1:64 prescale value
100 = 1:32 prescale value
011 = 1:16 prescale value
010 = 1:8 prescale value
001 = 1:4 prescale value
000 = 1:2 prescale value

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared x = Bit is unknown

Figura 30. Configuração dos bits do T0CON. (DATA, 2006).

Quando o Timer0 ultrapassa sua contagem máxima, seu *flag* (bit) de interrupção é acionado (TMR0IF=1) no registrador que controla a interrupção deste periférico (INTCON). Se o bit TMR0IE do INTCON estiver habilitado (TMR0IE=1), e suas interrupções estiverem habilitadas através do bit GIE (GIE=1), uma interrupção é gerada.

As funções dos bits do INTCON são apresentadas na Figura 31:

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts
When IPEN = 1:
1 = Enables all high priority interrupts
0 = Disables all interrupts

bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts
When IPEN = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE:** INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt
0 = Disables the INT0 external interrupt

bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INT0IF:** INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software)
0 = The INT0 external interrupt did not occur

bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Note: A mismatch condition will continue to set this bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared.

Figura 31. Configuração dos bits do INTCON. (DATA, 2006).

A configuração do TIMER0 para que ele funcione como um temporizador de 8 bits deve seguir as etapas:

1. Zerar o bit T0CS do registrador T0CON para selecionar o *clock* de $F_{clkprocessdor}/4$;

2. Usar os bits T0PS2:T0PS0 do registrador T0CON para selecionar o valor de pré-escala;

3. Zerar o bit PSA do registrador T0CON para selecionar o modo pré-escala;

4. Carregar o valor de temporização no registrador TMR0L de acordo com a seguinte fórmula:

$Overflow\ time\ = 4\ \times Tosc \times Prescaler\ \times (256 -\ \ TMR0)$, (1)

onde: Overflow time é o tempo em µs para ocorrer o overflow; Tosc é o período em µs de oscilação do sinal de clock de entrada, ou seja, 1/ $F_{clkprocessdor}$; Prescaler é o valor da pré-escala; e TMR0 é o valor do byte a ser carregado no registrador TMR0L.

No modo de 16 bits, o valor a ser atribuído na variável TMR0 de (1) é dividido em dois registradores: TMR0L e TMR0. O byte mais significativo deve ser carregado no registrador TMR0 e o menos significativo no TMR0L. A constante 256 de (1) deve ser substituída pelo valor 65536 para se operar o temporizador com 16 bits.

7.2.2.Displays de 7 segmentos

Segundo (GUIA, 2001), a placa possui quatro displays de 7 segmentos, sendo que todos os 8 leds necessários para formar um dígito (7 segmentos mais o ponto), de cada um dos 4 displays, estão conectados simultaneamente ao PORTD, conforme ilustra a Figura 32.

DISPLAY'S DE 7 SEGMENTOS

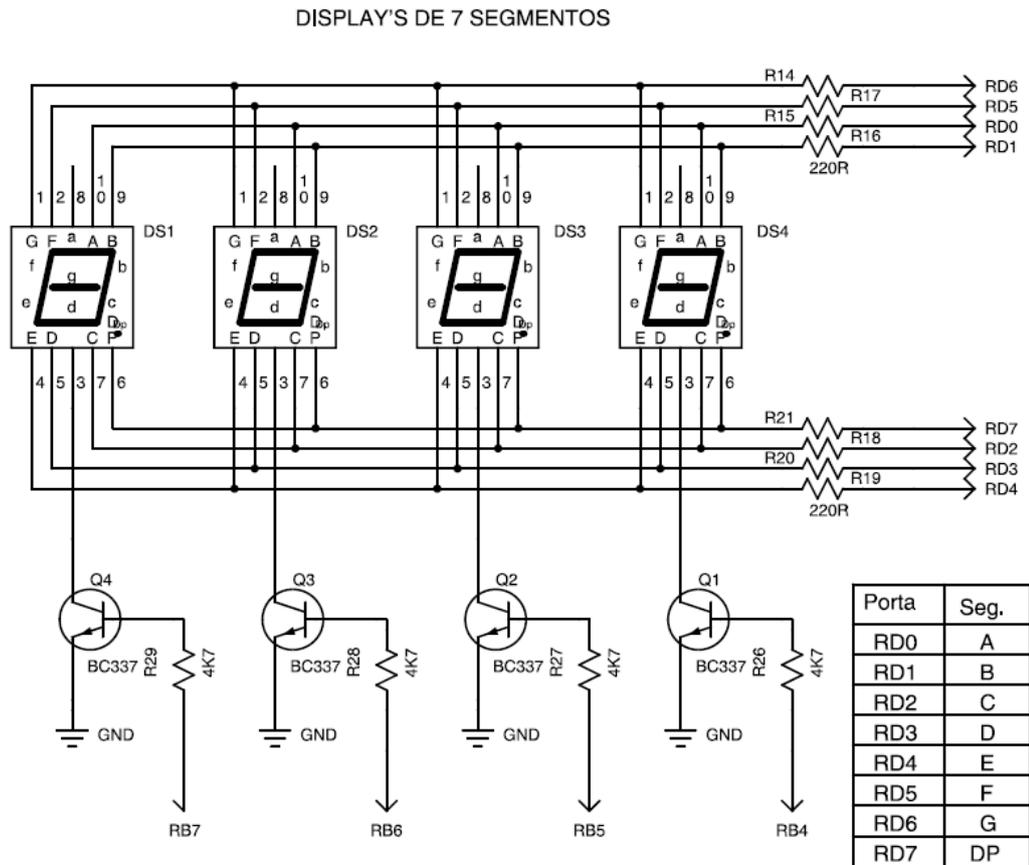



Figura 32. Esquema elétrico do display de 7 segmentos da placa McLab2. (GUIA, 2001).

As vias de seleção de cada um dos displays é feita através dos 4 bits mais significativos do PORTB e seguem a Tabela 14:

Tabela 14. Pinagem do PORTB referente aos displays de 7 segmentos da placa McLab2. (GUIA, 2001).

PIC	*Display*
RB7	DS1 (milhar)
RB6	DS2 (centena)
RB5	DS3 (dezena)
RB4	DS4 (unidade)

Para que os leds dos displays sejam acesos, deve-se escrever o valor 1 no bit do PORTD (RD0 a RD7) referente ao led que se quer acender e no bit do PORTB

(RB4 A RB7) do display que se quer acionar. Observe na Figura 32 que todos os segmentos dos displays estão ligados simultaneamente ao PORTD, logo se todos os displays forem acionados ao mesmo tempo, o mesmo dígito aparecerá.

7.2.3.Teclado

Conforme (GUIA, 2001), existem 4 teclas na placa. Quando elas estão em estado normal (normalmente aberto), o microcontrolador deverá ler nível lógico 1 nas portas do teclado. Quando uma tecla é pressionada, o nível lógico presente na porta do microcontrolador passa a 0. As teclas estão conectadas ao microcontrolador através dos 4 bits menos significativos da PORTB, conforme ilustra a Figura 33:

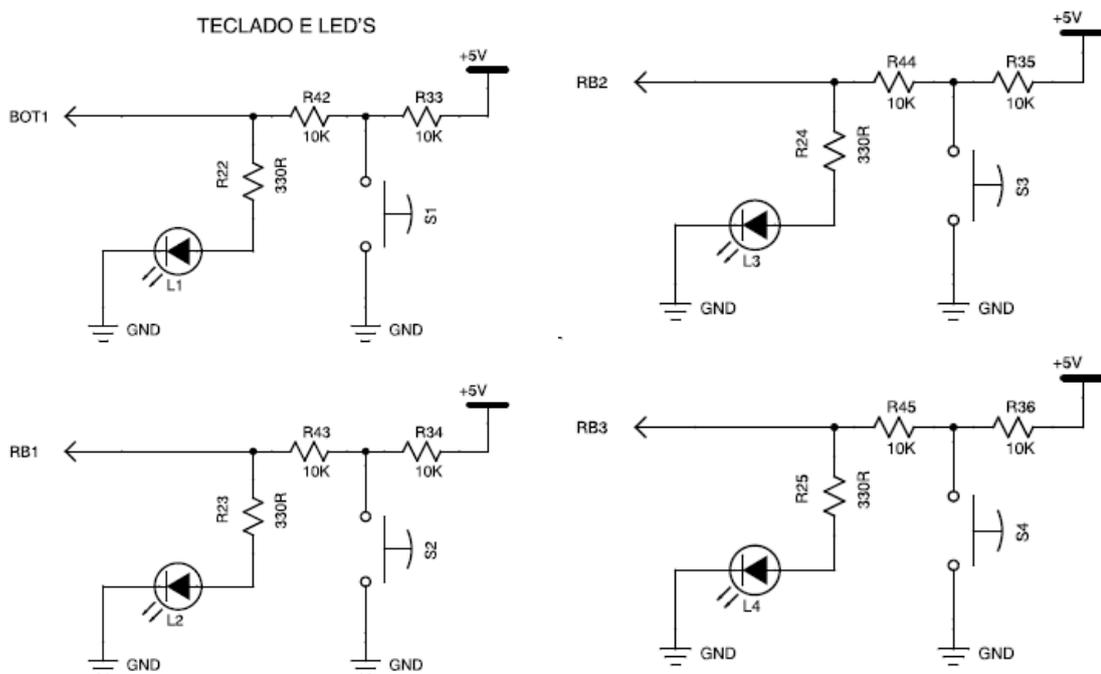



Figura 33. Esquema elétrico das teclas da placa McLab2. (GUIA, 2001).

A distribuição da pinagem do PORTB para cada tecla da placa segue a Tabela 15.

Tabela 15. Pinagem do PORTB referente às teclas da placa McLab2. (GUIA, 2001).

PIC	*Tecla*
RB0	S1
RB1	S2
RB2	S3
RB3	S4

7.3.Texto descritivo da aplicação

A aplicação utilizada para a análise comparativa entre os programa com, e sem, o RTOS é um contador crescente/decrescente de 4 dígitos que possui os seguintes requisitos:

- Os 4 dígitos deverão ser exibidos no display de 7 segmentos;
- A contagem deverá ter incremento ou decremento a 10 Hz ou 100 ms;
- O acionamento da contagem deve ser via Teclado e seguir as seguintes especificações:
    - Contagem crescente: tecla S1 pressionada;
    - Contagem decrescente: tecla S2 pressionada;
    - Parar contagem: tecla S3 pressionada;
    - Zerar contagem: tecla S4 pressionada.

Para deixar mais claro o comportamento desejado, um diagrama de estados é mostrado na Figura 34:

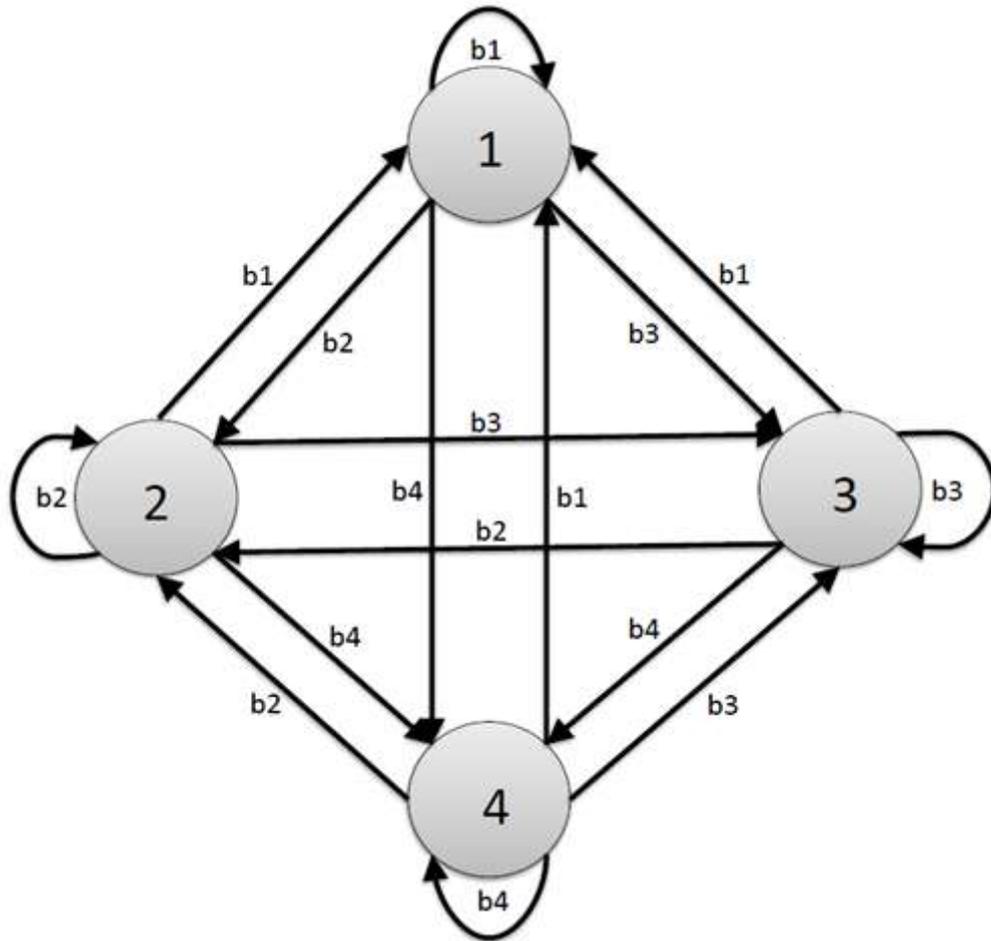



Figura 34. Diagrama de estados do contador crescente/decrescente.

Nesta máquina de estados, cada círculo numerado representa um estado. A alteração de um estado para outro se dá através do pressionamento das teclas (sinais b1, b2, b3, b4), sendo que cada tecla pertence a um estado diferente:

- Tecla S1 (sinal b1) → Estado 1: Contagem crescente;
- Tecla S2 (sinal b2) → Estado 2: Contagem decrescente;
- Tecla S3 (sinal b3) → Estado 3: Contagem parada;
- Tecla S4 (sinal b4) → Estado 4: Contagem zerada.

As flechas indicam a mudança de um estado para outro quando uma tecla é pressionada. Por exemplo, se a máquina estiver no estado 1, ela está contando crescentemente. Quando a tecla S2 é pressionada, o estado da máquina deve ser

alterado para o estado 2 e a contagem deve ser alterada para decrescente. Independente de qual estado a máquina estiver, seu estado deve ser alterado quando uma tecla for pressionada, a menos que a tecla pressionada seja a relacionada ao estado presente.

A Figura 35 ilustra o diagrama de partição do contador, separando-os nos seguintes grupos de funções: Entradas, Saídas, Temporizações e Aplicação.

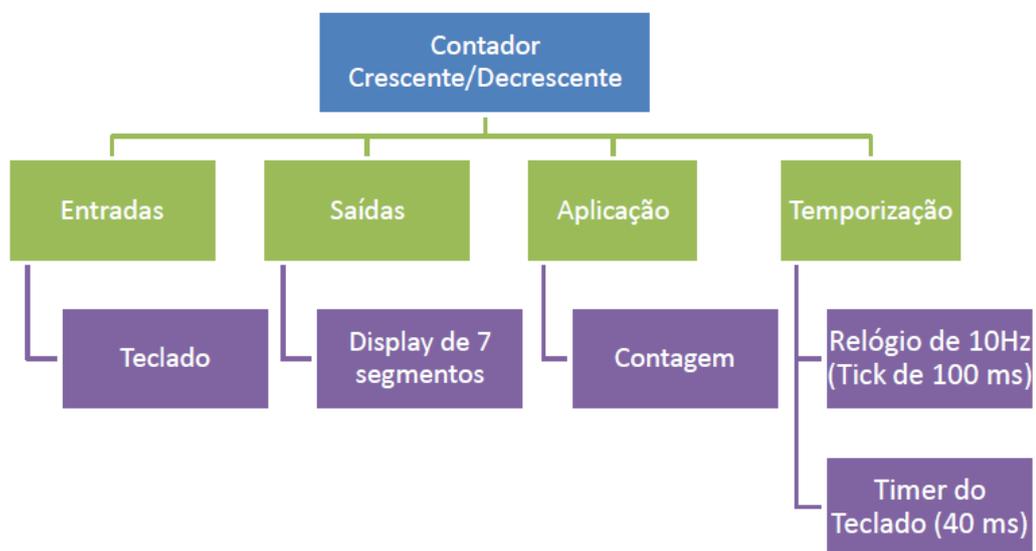



Figura 35. Diagrama de partição do contador crescente/decrescente.

A existência do bloco de funções Temporização deve-se aos requisitos de tempo impostos pela aplicação: o incremento da contagem a cada 100 ms, representado pelo processo Relógio de 10 Hz; e o debounce[7] das teclas, representado pelo processo Timer do Teclado. Estes requisitos de tempo classificam o contador crescente/decrescente como um sistema de tempo real.

[7] O problema do debounce ocorre ao se pressionar uma tecla. No momento do pressionamento da tecla, pulsos de ruído são gerados na entrada do MCU relativa àquela tecla, fazendo com que este possa ler níveis lógicos alternantes. O software presente no MCU pode interpretar que aquela tecla, pressionada uma única vez, foi pressionada mais vezes, podendo corromper o sistema.

Existe mais um requisito de tempo que está oculto no bloco do display de 7 segmentos: a multiplexação de cada display no tempo para que cada um mostre o seu dígito correspondente corretamente. Este requisito de tempo deve-se ao hardware utilizado, conforme descrito na seção 7.2.2. Para que cada display aparente estar constantemente aceso e mostre o dígito correto, é necessário acioná-lo a cada 20 ms. Caso este tempo seja maior, haverá cintilação do mesmo. Caso seja menor, os dígitos não ficarão bem definidos no display.

Exceder os limites de tempo impostos pelo contador não causam problemas graves aos usuários do mesmo, a não ser diminuir a precisão da contagem ou não mostrá-la corretamente. Logo, podemos classificar esta aplicação, como um sistema de tempo real do tipo brando, conforme definição na seção 3.1.

O desenvolvimento dos softwares a serem comparados, para se avaliar o uso do RTOS, foi baseado no experimento criado pelo Professor Doutor Luis Otávio Saraiva Ferreira (link: http://www.fem.unicamp.br/~lotavio/) para a aula Nº08 da disciplina ES770 – Laboratório de Sistemas Digitais. Esta disciplina foi oferecida no primeiro semestre de 2009, pelo curso de Engenharia de Controle e Automação da Universidade Estadual de Campinas. O experimento, na íntegra, se encontra em ANEXO.

7.4. Critérios utilizados para a comparação entre os softwares

Os critérios que serão utilizados para se comparar os dois softwares serão:

- Precisão da contagem no incremento e decremento da mesma, medida através da ferramenta Stopwatch do software MPLAB® IDE;

- Facilidade da codificação do programa, medida através da:
    - Quantidade de máquinas de estado criadas pelo programador;
    - Quantidade de estados criados pelo programador;
    - Quantidade de sinais criados pelo programador.

7.5.Contador crescente/decrescente sem RTOS

O experimento localizado em ANEXO mostra o desenvolvimento do contador crescente/decrescente com apenas três estados: Contagem Crescente, Contagem Decrescente e Parar Contagem. No final do experimento é proposto, como exercício, incluir o quarto estado: Zerar Contagem. Com estes quatro estados e com o incremento/decremento da contagem a 10 Hz (ao invés de 4 Hz), o contador desenvolvido no experimento em ANEXO se equivale ao proposto neste trabalho.

O código em linguagem C do programa do contador sem o uso do sistema operacional de tempo real se encontra no APÊNDICE B.

7.5.1.Diagrama de interação entre blocos do contador

A inclusão do quarto estado do contador adiciona ao diagrama de interação entre blocos do experimento em ANEXO um sinal de entrada no teclado (s4), e um sinal enviado para a contagem (ZERA ou b4), que representa o estado do novo sinal de entrada do teclado. A Figura 36 mostra o novo diagrama de interação entre blocos do contador:

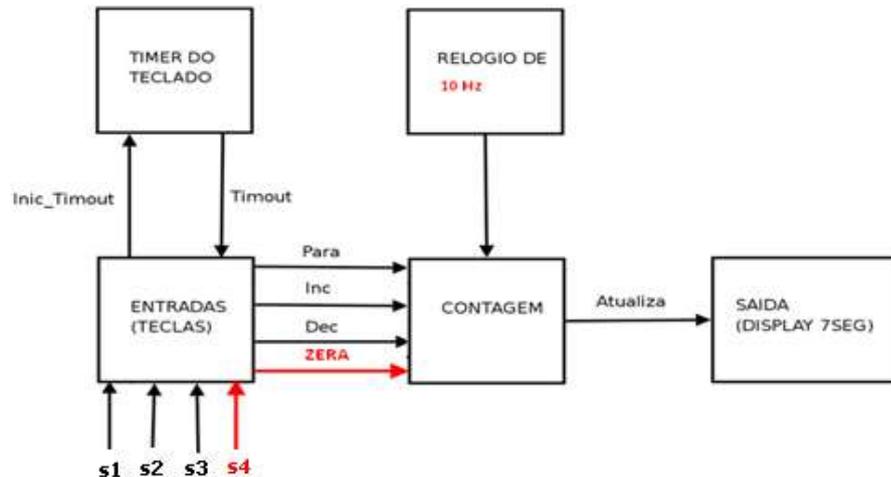



Figura 36. Diagrama de interação entre os blocos funcionais do contador com a inclusão do estado “zerar contagem”.

7.5.2.O processo TECLADO

7.5.2.1.Descrição do processo TECLADO

O processo TECLADO é uma máquina de dois estados: ESPERA_TECLA e ESPERA_TIMOUT. Quando está no estado ESPERA_TECLA, o processo verifica se uma tecla é pressionada, aguardando receber um dos sinais s1, s2, s3 ou s4. Caso uma tecla seja pressionada, envia o sinal Sinal_Inic_Timout para o processo TIMER_TECLADO, onde será feito a temporização de 40 ms para tratar do debounce das teclas, e altera seu estado para ESPERA_TIMOUT. Neste estado, aguarda o sinal Sinal_Timout_Teclado do processo TIMER_TECLADO. Quando recebe o sinal Sinal_Timout_Teclado, confirma se a tecla continua pressionada ou não. Caso a tecla esteja pressionada, reconhece qual tecla foi pressionada, e envia o sinal correspondente à tecla pressionada para o processo CONTADOR, através dos sinais b1, b2, b3 ou b4, e retorna para o estado ESPERA_TECLA. Caso não

haja pressionamento das teclas, a máquina permanece no estado ESPERA_TECLA ou vai para este se estiver no estado ESPERA_TIMOUT.

A Figura 37 apresenta o diagrama SDL do processo TECLAS (TECLADO). Neste diagrama está representada a dinâmica de funcionamento do teclado como máquina de estado.

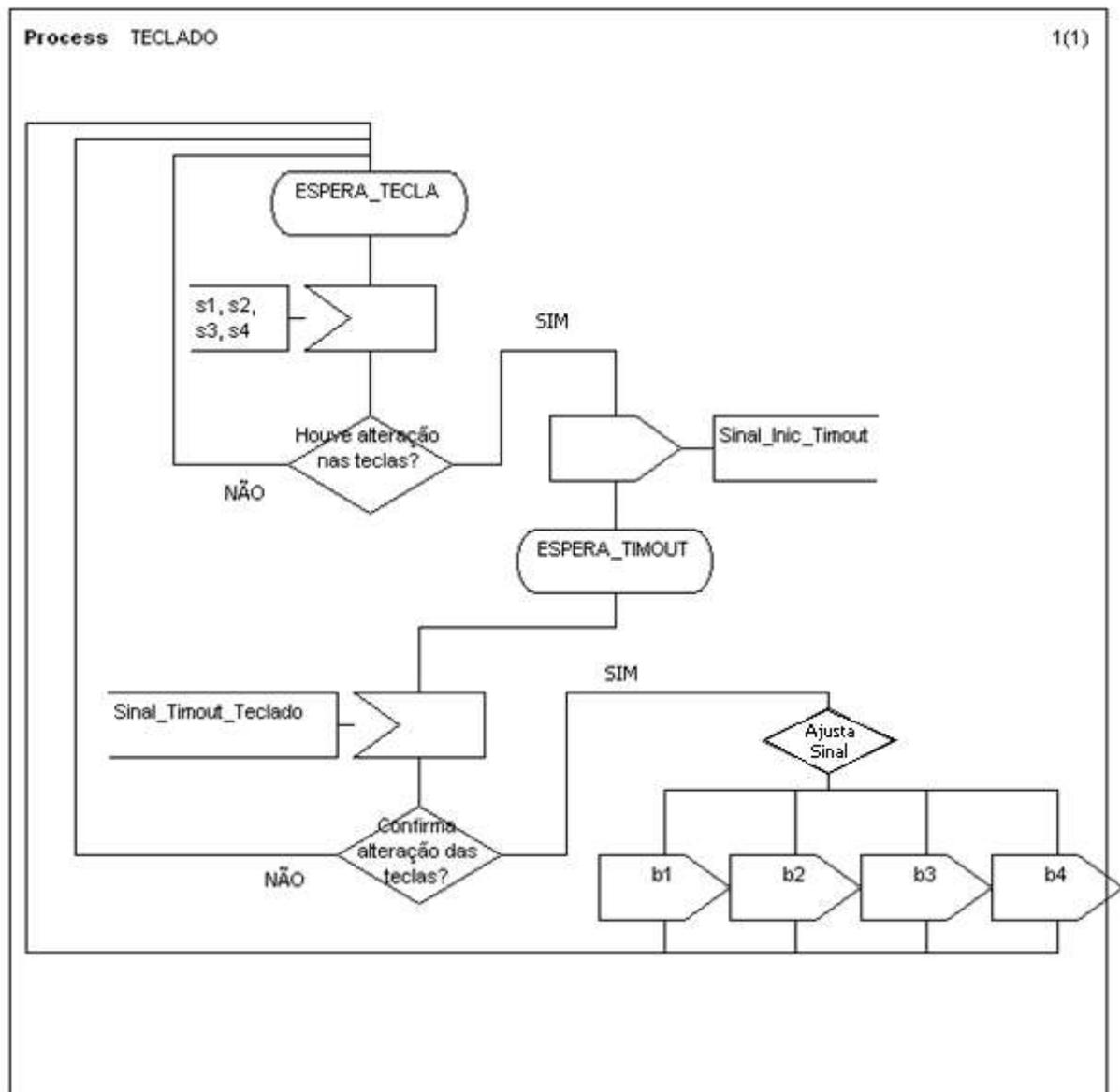



Figura 37. Diagrama SDL do TECLADO.

7.5.2.2.Características do processo TECLADO

- Estados:
    - ESPERA_TECLA;
    - ESPERA_TIMOUT;
- Sinais de Entrada:
    - Sinal_Timout_Teclado (vem do processo Timer_TECLADO);
    - Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4 (Leitura das teclas do teclado);
- Sinais de Saída:
    - Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4 (vai para o processo CONTADOR);
    - Sinal_Inic_Timout (vai para o processo Timer_TECLADO);
- Entrada de dados:
    - NENHUM;
- Saída de dados:
    - NENHUM.

7.5.3.O processo CONTADOR

7.5.3.1.Descrição do processo CONTADOR

O processo CONTADOR pode se encontrar em cada um dos seguintes estados: ou está no estado crescente, onde incrementa a contagem; ou está no estado decrescente, onde decrementa a contagem; ou está parado; ou está zerado. Quando recebe o sinal Sinal_TIC_100, do processo TIMER_100, executa a ação

correspondente ao seu estado, envia o NUMERO (por intermédio da função auxiliar atualiza_display – ver APÊNDICE B) para o processo MaqDisplayLEDs e volta ao início do processo. Quando recebe os sinais b1, b2, b3 ou b4, altera o seu estado de contagem de acordo com o estado recebido.

O processo de contagem (CONTADOR) está representado com mais detalhes no diagrama SDL da Figura 38.

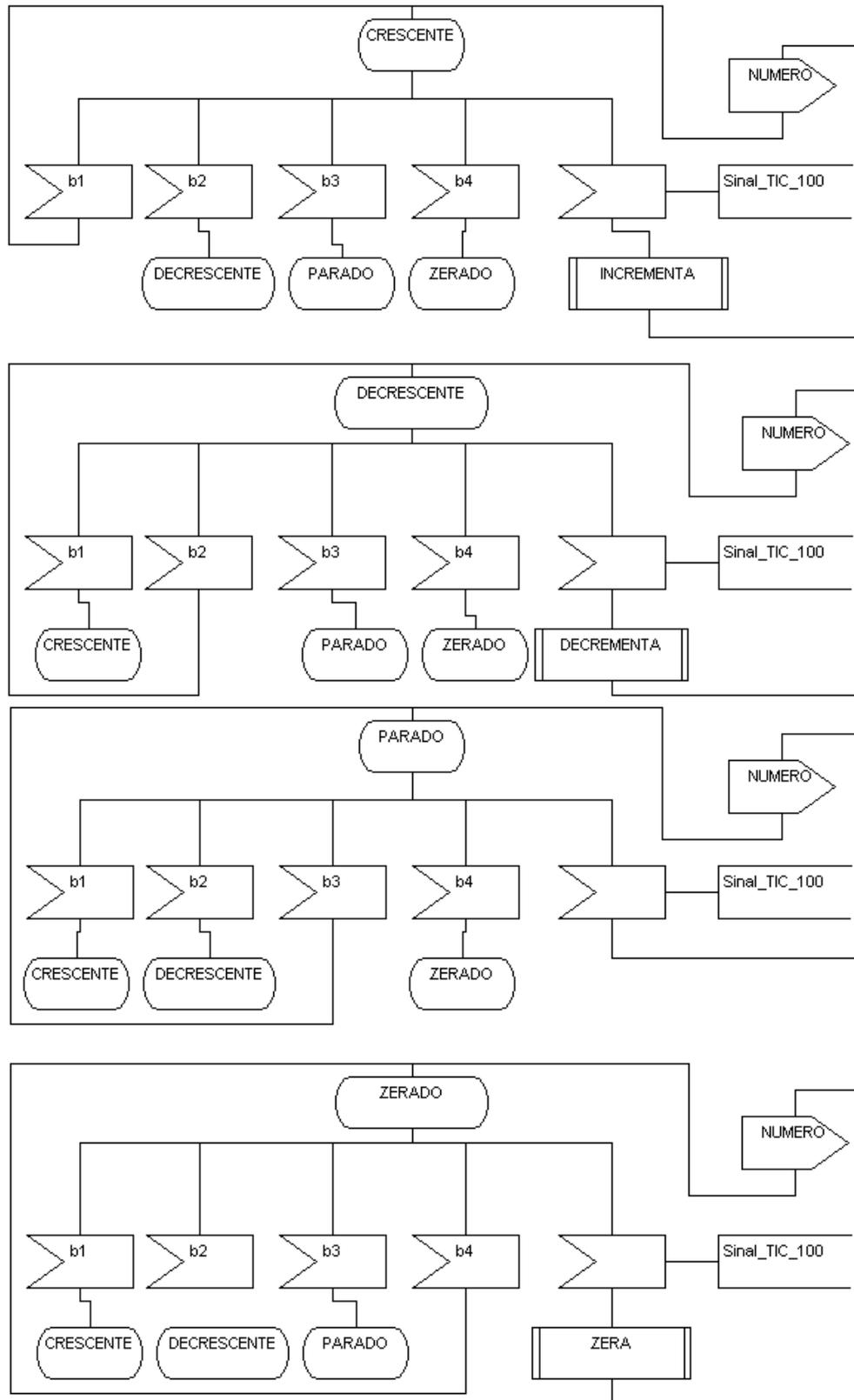



Figura 38. Diagrama SDL do processo CONTADOR.

7.5.3.2.Características do processo CONTADOR

- Estados:
    - PARADO;
    - CRESCENTE;
    - DECRESCENTE**;**
    - ZERO;
- Sinais de Entrada:
    - Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4 (vem do processo TECLADO);
    - Sinal_TIC_100 (vem do processo TIMER_100);
- Sinais de Saída:
    - NENHUM;
- Entrada de dados:
    - NENHUM;
- Saída de dados:
    - NUMERO (vai para o processo MaqDisplayLEDs) - Número de 16 bits que será mostrado no display.

7.5.4.O processo MaqDisplayLEDs

7.5.4.1.Descrição do processo MaqDisplayLEDs

O processo MaqDisplayLEDs mostra o valor da contagem no display de LEDs. Para isso, ele seleciona o dígito que será aceso alterando seqüencialmente seu próprio estado. Tem 4 estados, um para cada dígito usado no display de LEDs. Recebe o dígito do NUMERO a ser escrito no display (do processo CONTADOR)

relacionado ao estado correspondente, e chama a função auxiliar escreve_display_LEDs (ver APÊNDICE B) que escreve neste dígito, alterando o estado presente para o estado do dígito seguinte.

O processo Display 7Seg (MaqDisplayLEDs) tem seu diagrama SDL representado na Figura 39.

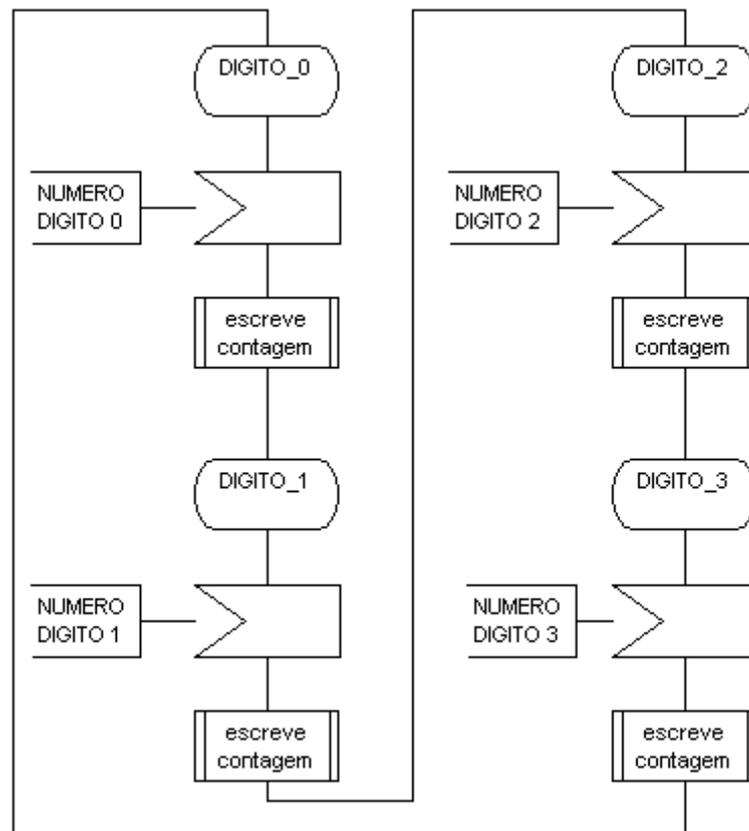



Figura 39. Diagrama SDL do processo MaqDisplayLEDS.

7.5.4.2.Características do processo MaqDisplayLEDs

- Estados:
    - DIGITO_3;
    - DIGITO_2;
    - DIGITO_1;

- DIGITO_0;
- Entradas de dados:
    - Digito do NUMERO do estado presente (vem do processo CONTADOR);
- Saída de dados:
    - NENHUMA;
- Sinais de Entrada:
    - NENHUM;
- Sinais de Saída:
    - NENHUM.

7.5.5.O processo TIMER_TECLADO

7.5.5.1.Descrição do processo TIMER_TECLADO

Esta máquina de estados realiza a temporização de 40 ms da eliminação do repique do teclado (debounce). Tem dois estados: TIMER_TECL_PARADO e TIMER_TECL_CONTANDO. Quando está no estado TIMER_TECL_PARADO, a máquina aguarda o sinal Sinal_Inic_Timout , para iniciar a temporização de 40ms. Ao receber o sinal Sinal_Inic_Timout, a variável que faz a contagem do tempo é inicializada e a máquina muda seu estado para o estado TIMER_TECL_CONTANDO. Neste estado o contador é decrementado até zero. Quando o contador atinge o valor zero, emite o sinal de Sinal_Timout_Teclado para o processo do TECLADO, voltando ao estado TIMER_TECL_PARADO.

O diagrama SDL do processo TIMER_TECLADO é representado na Figura 40.

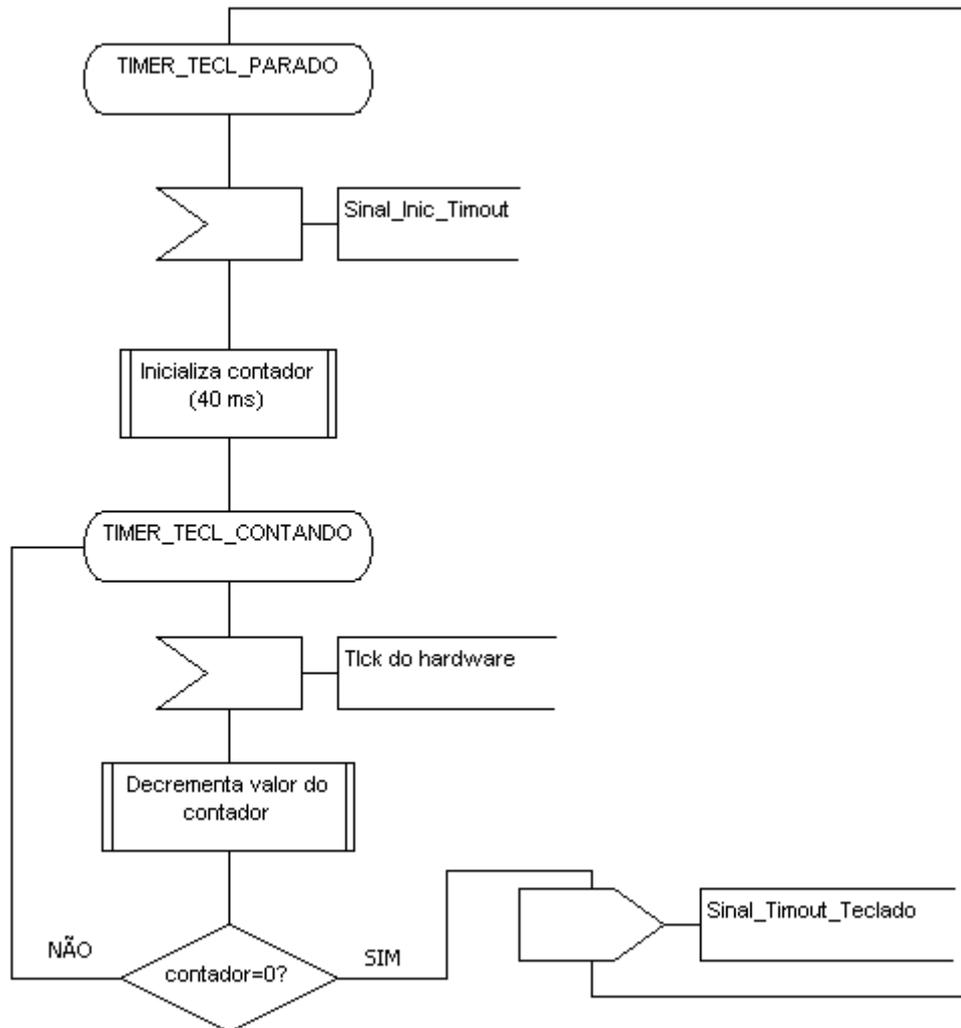



Figura 40. Diagrama SDL do processo TIMER_TECLADO.

7.5.5.2.Características do processo TIMER_TECLADO

- Estados:
  - TIMER_TECL_PARADO;
  - TIMER_TECL_CONTANDO;
- Entradas de dados:
  - NENHUMA;
- Saídas de dados:
  - NENHUMA;

- Sinais de Entrada:
    - Sinal_Inic_Timout (vem do processo TECLADO);
- Sinais de Saída:
    - Sinal_Timout_Teclado (vai para o processo TECLADO).

7.5.6.O processo TIMER_100

7.5.6.1.Descrição do processo TIMER_100:

O processo RELÓGIO DE 10 Hz (TIMER_100) temporiza o intervalo de 100 milissegundos entre incrementos ou decrementos do contador. Para isto, ele decrementa um contador a cada *tick* do hardware até que a contagem chegue à zero. Quando o contador atinge o valor zero, este é reiniciado e a contagem recomeça.

O processo TIMER_100 tem seu diagrama SDL representado na Figura 41.

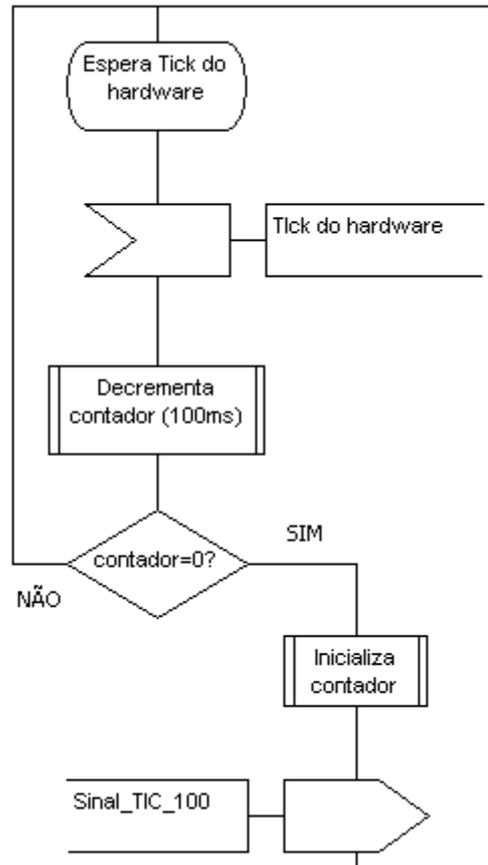



Figura 41. Diagrama SDL do TIMER_100.

7.5.6.2.Características do processo TIMER_100

- Estados:
  - Único;
- Entradas de dados:
  - NENHUMA;
- Saída de dados:
  - NENHUMA;
- Sinais de Entrada:
  - NENHUM;
- Sinais de Saída:

- Sinal_TIC_100 (vai para o processo CONTADOR).

7.5.7.O escalonador cíclico e seu intervalo de interrupção

Para que o programa tenha comportamento de tempo real é necessário fazer uma rotina que gerencie a distribuição do tempo do processador aos processos. Esta rotina, como nos sistemas operacionais, é denominada de Escalonador. Um escalonador, que não possui as ferramentas de um sistema operacional para ser implementado, passa o controle da CPU aos processos, de forma seqüencial e ordenada, num intervalo de tempo fixo. Este tipo de escalonador é denominado de escalonador cíclico.

A Figura 42 ilustra o fluxograma do escalonador cíclico implementado no contador crescente/decrescente. A cada interrupção do TIMER0, todos os processos são invocados para execução na ordem da figura.

O valor do intervalo de interrupção, ou seja, o tempo entre duas interrupções consecutivas, foi escolhido em função do Display de 7 segmentos, devido a necessidade de se fazer a multiplexação dos displays no tempo, conforme informado na seção 7.3. Como cada display deve ter um período de acionamento de 20 ms, a máquina de estado MaqDisplayLEDs deve alterar seu estado a cada 5 ms, já que são 4 displays. Portanto o intervalo de interrupção do TIMER0 a ser utilizado será de 5 ms.

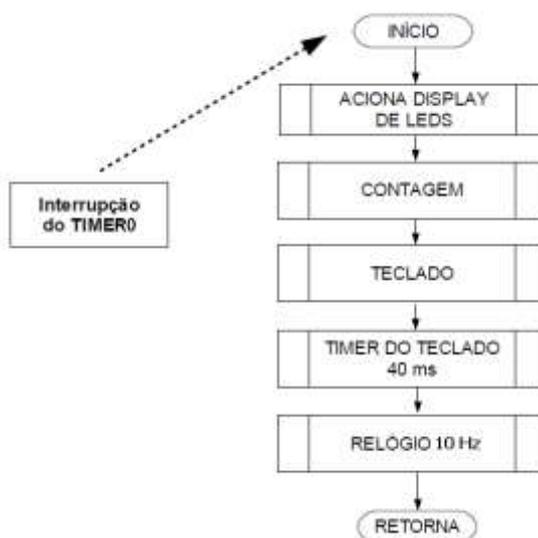



Figura 42. Fluxograma do Escalonador Cíclico.

7.6.Contador crescente/decrescente com RTOS

A principal diferença de se utilizar o sistema operacional de tempo real (RTOS) reside no fato de que o kernel tratará de toda a temporização do sistema, sem que haja a necessidade do programador se preocupar em desenvolver processos para tratar destas temporizações. A única preocupação que o programador terá é saber utilizar os serviços oferecidos pelo RTOS para que esta temporização seja alcançada de acordo com sua necessidade. Portanto, no caso do contador crescente/decrescente aqui descrito, os processos de temporização TIMER_TECLADO e TIMER_100 serão desconsiderados. O código do programa do contador com o uso do sistema operacional de tempo real se encontra no APÊNDICE A.

Os processos CONTADOR e TECLADO foram tratados como tarefas (ver seção 5.6.1) do sistema operacional de tempo real e o processo MaqDisplayLEDs foi

tratado como uma rotina chamada a cada interrupção do TIMER0. Cada processo será tratado em detalhes nesta seção.

A Figura 43 ilustra a interação do contador com o ambiente externo e com o kernel. O kernel é tratado como um sistema que coordena as atividades de outro sistema, no caso o contador. Uma análise detalhada de como o kernel do µC/OS-II trata as tarefas é representada na Figura 26. Os sinais recebidos pelo kernel determinam em quais estados estarão as atividades (tarefas) da aplicação.

Para o contador, foram utilizados os seguintes serviços do sistema operacional:

- 1 semáforo (ver seção 5.6.2) – necessário para sincronizar as duas tarefas, para que o processo CONTADOR receba corretamente a mensagem do processo TECLADO;
- 1 caixa de correio (ver seção5.6.3) – que conterá a mensagem que informará ao processo CONTADOR o seu estado, conforme o estado das teclas;
- Serviço de temporização OSTimeDly() para a execução das tarefas nos requisitos de tempo da aplicação.
- OSSemPend() e OSSemPost() para operar sobre o semáforo;
- OSMboxPend() e OSMboxPost() para operar sobre a caixa de correio.

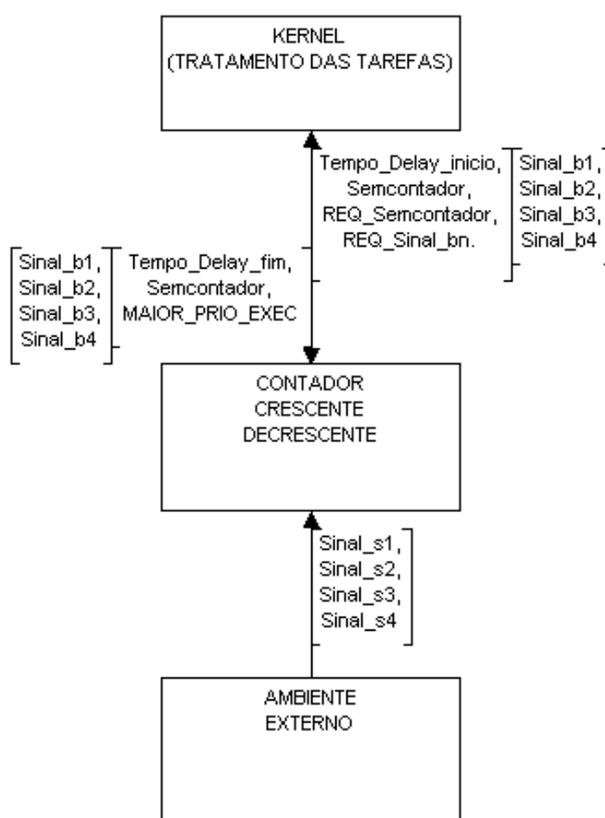



Figura 43. Ilustração da interação entre o contador, o kernel e o ambiente externo.

Os sinais trocados pelos sistemas da Figura 43 possuem os seguintes significados:

- **Tempo_Delay_inicio -** sinal enviado pelas tarefas ao kernel para requisitar a sua suspensão por um tempo determinado, ou seja, para que o kernel mude os estados das tarefas, do estado “em execução” para o estado “bloqueado”, pela quantidade de tempo requisitada por elas. No caso do µC/OS-II, este sinal é pertencente ao serviço OSTimeDly().
- **Tempo_Delay_fim** - sinal enviado pelo kernel às tarefas, informando que o tempo de suspensão requisitado por elas se esgotou. Dessa forma o kernel passa o estado das tarefas do estado “bloqueado” para o estado “pronto para execução”.

- **MAIOR_PRIO_EXEC** - sinal enviado pelo kernel às tarefas para que elas sejam executadas, ou seja, mudando seu estado de “pronto para execução” para “em execução”.
- **Semcontador** – sinal trocado entre a tarefa CONTADOR, o kernel e a tarefa TECLADO, utilizado para a confirmação da tarefa CONTADOR sobre o recebimento da mensagem enviada pela tarefa TECLADO. No caso do µC/OS-II, este sinal é enviado pelo CONTADOR (sentido: CONTADOR → kernel) através do serviço OSSemPost() e é recebido pelo TECLADO (sentido: kernel → TECLADO) através do serviço OSSemPend();
- **REQ_Semcontado**r – sinal enviado da tarefa TECLADO ao kernel para requisitar a confirmação do recebimento da mensagem enviada para a tarefa CONTADOR. No caso do µC/OS-II, este sinal pertence ao serviço OSSemPend();
- **Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4** – sinais contendo as mensagens, sobre o estado do processo CONTADOR, trocadas entre a tarefa CONTADOR e a tarefa TECLADO, intermediadas pelo kernel (analogia a um carteiro). No µC/OS-II, estes sinais são enviados pelo TECLADO (sentido: TECLADO → kernel) através do serviço OSMboxPost() e recebidos pelo CONTADOR (sentido: kernel → CONTADOR) através do serviço OSMboxPend();
- **REQ_Sinal_bn** – requisição feita pela tarefa CONTADOR ao kernel para que este lhe entregue a mensagem enviada pela tarefa TECLADO. Este sinal é pertencente ao serviço OSMboxPend() do µC/OS-II.
- **Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4 –** sinais enviados ao processo TECLADO que indica o estado das teclas.

7.6.1.Diagrama da interação entre blocos do contador

A Figura 44 ilustra, com detalhes, as interações entre os blocos funcionais interiores ao contador crescente/decrescente. Perceba que não há interação direta entre as tarefas TECLADO e CONTADOR, pois toda comunicação dos sinais e dados entre estas tarefas deve ser intermediada pelo kernel. Perceba também que os processos para a temporização (TIMER DO TECLADO e RELÓGIO DE 10Hz) não existem, pois o kernel trata desta temporização com um de seus serviços de gerenciamento do tempo que é invocado pela tarefa requisitante. No caso do µC/OS-II, o serviço utilizado é o OSTimeDly(),

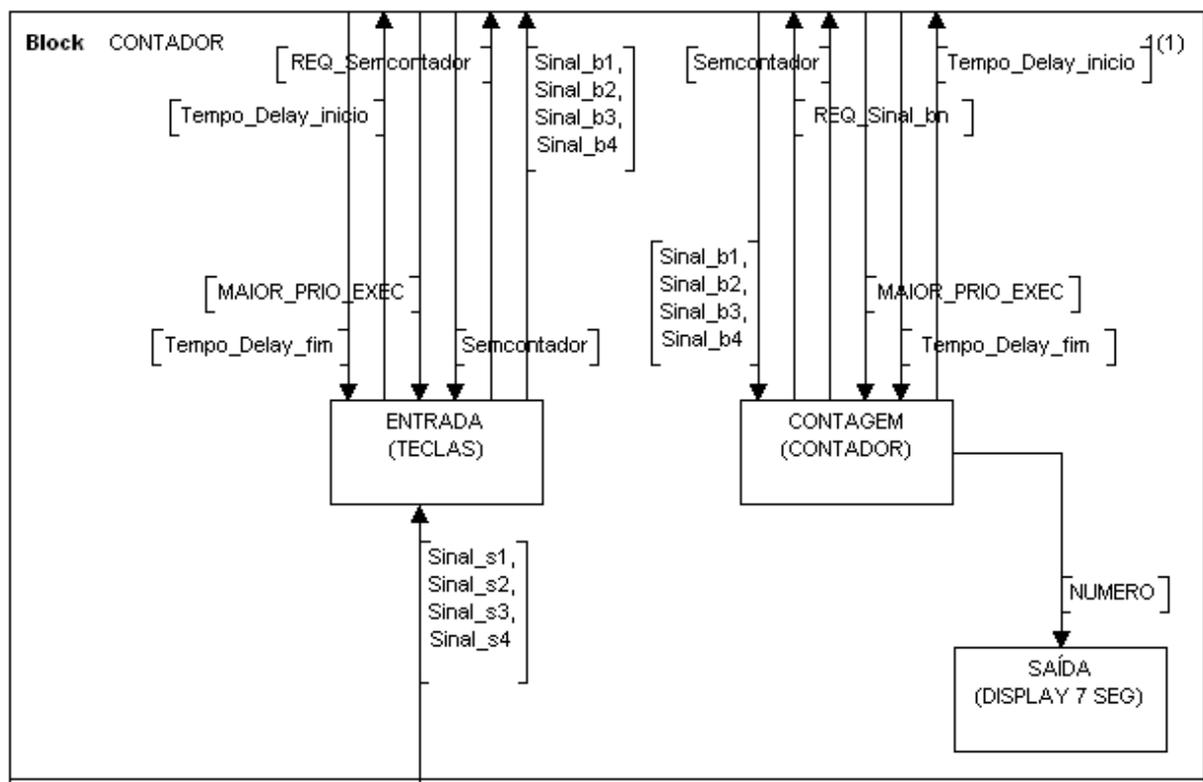



Figura 44. Diagrama de interação entre blocos funcionais do contador.

O programador que utiliza um sistema operacional de tempo real, não manipula todos os sinais trocados entre as tarefas e o kernel, pois alguns sinais são intrínsecos aos serviços do mesmo. Dessa forma, o programador não necessita trabalhar diretamente com estes sinais, bastando apenas utilizar os serviços providos pelo kernel. Um exemplo desta situação são os sinais Tempo_Delay_inicio e Tempo_Delay_fim. Ao invés de trabalhar diretamente com estes sinais, o programador apenas invoca o serviço OSTimeDly() e o kernel fará todo o trabalho necessário para tratar do atraso da execução da tarefa que requisitou este serviço. Os sinais que o programador deve manipular são os denominados eventos ou objetos do kernel (ver seção 5.6), que, no caso desta aplicação, são o semáforo e a caixa de correio. O diagrama de interação entre blocos funcionais do contador, na "visão" do programador, é ilustrado na Figura 45.

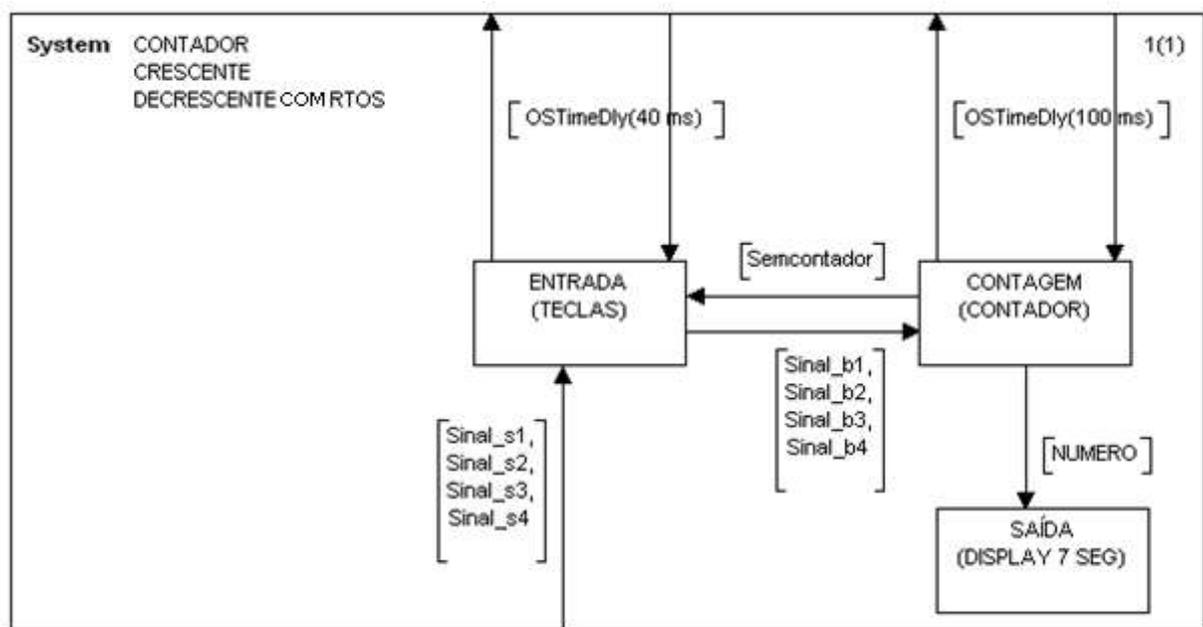



Figura 45. Diagrama de interação entre blocos funcionais do contador na "visão" do programador.

7.6.2.A tarefa TECLADO

7.6.2.1.Descrição da tarefa TECLADO

A tarefa TECLADO lê o estado das teclas e verifica se houve alteração no seu estado. Se houve alteração, espera por 40 ms para tratar do debounce, chamando um serviço de gerenciamento do tempo, no caso o OSTimeDly(). Passado o tempo de espera requisitado, esta tarefa verifica se a tecla continua pressionada ou não. Caso haja confirmação da alteração das teclas, reconhece qual tecla foi pressionada (sinais s1, s2, s3 ou s4) e envia uma mensagem (sinais b1, b2, b3, b4) para a tarefa CONTADOR através do serviço OSMboxPost(), contendo o estado correspondente a tecla pressionada. Se não houver alteração das teclas, envia uma mensagem para a tarefa CONTADOR, contendo a informação de que não houve alteração das teclas. Após o envio da mensagem, a tarefa TECLADO aguarda a confirmação de que a tarefa CONTADOR recebeu a mensagem enviada, ao requisitar o semáforo (REQ_Semcontador) através do serviço OSSemPend(). Quando recebe este semáforo (Semcontador) a tarefa reinicia uma nova verificação das teclas. A tarefa teclado possui a menor prioridade entre as duas tarefas do sistema.

O diagrama SDL apresentado na Figura 46 para o processo TECLADO, agora com o uso do RTOS, inclui os estados das tarefas para mostrá-lo como tal. Repare que o processo TECLADO não possui estados definidos pelo programador. Todos os estados descritos são referentes às tarefas e é responsabilidade do kernel alterá-los de acordo com os serviços utilizados. Todos os sinais trocados com o kernel e também com outros processos estão representados neste diagrama.

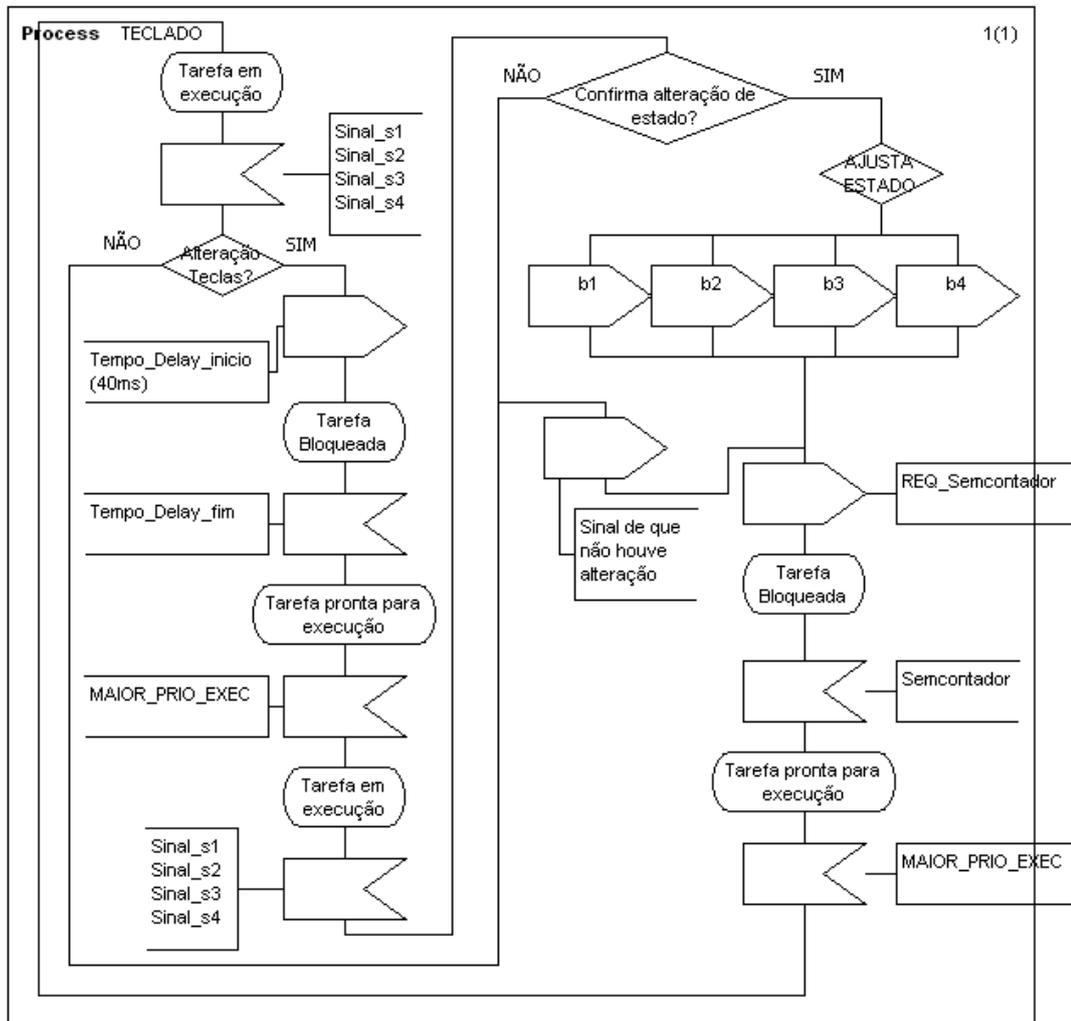



Figura 46. Diagrama SDL do processo TECLADO com o uso de um RTOS.

7.6.2.2.Características da tarefa TECLADO

- Estados:
    - Tarefa "em execução" – estado determinado pelo kernel;
    - Tarefa "pronta"– estado determinado pelo kernel;
    - Tarefa "bloqueada" – estado determinado pelo kernel;
- Entradas de dados:
    - NENHUMA;
- Saidas de dados:

- NENHUMA;
- Sinais de Entrada:
  - Semcontador (vem do processo CONTADOR). Utiliza o serviço OSSemPend();
  - Tempo_Delay_fim (vem do kernel) – sinal determinado pelo kernel;
  - MAIOR_PRIO_EXEC (vem do kernel) – sinal determinado pelo kernel;
  - Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4 (Leitura das teclas do teclado);
- Sinais de Saída:
  - Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4 (vai para o processo CONTADOR). Utiliza o serviço OSMboxPost();
  - Sinal de não alteração das teclas. Utiliza o serviço OSMboxPost();
  - Tempo_Delay_inicio (vai para o kernel) - sinal determinado pelo kernel. Pertencente ao serviço OSTimeDly;
  - REQ_Semcontador (vai para o kernel) - sinal determinado pelo kernel. Pertencente ao serviço OSSemPend();

7.6.3.A tarefa CONTADOR

7.6.3.1.Descrição da tarefa CONTADOR

A tarefa CONTADOR aguarda receber a mensagem da tarefa TECLADO (sinais b1, b2, b3, ou b4) através do serviço OSMboxPend() para iniciar sua execução. Esta mensagem contém o estado da contagem da tarefa CONTADOR. Quando recebe esta mensagem, a tarefa CONTADOR envia um semáforo a tarefa TECLADO através do serviço OSSemPost() para confirmar o recebimento da

mensagem e executa a ação referente ao estado recebido: se o estado for crescente, incrementa a contagem; se for decrescente, decrementa a contagem; se for parado, não faz nada; ou se for zerado, zera a contagem. Após atualizar a contagem, o NUMERO é enviado para o processo MaqDisplayLEDs utilizando-se da função auxiliar atualiza_display (). Então, a tarefa CONTADOR invoca o serviço de tempo OSTimeDly() para parar sua execução por 100 ms. Esgotado o tempo, a tarefa CONTADOR reinicia sua execução. Esta é a tarefa de maior prioridade.

O processo contador tem como diagrama SDL, o apresentado na Figura 47. Como no diagrama SDL do processo TECLADO, neste diagrama também estão inclusos os estados e sinais determinados pelo kernel.

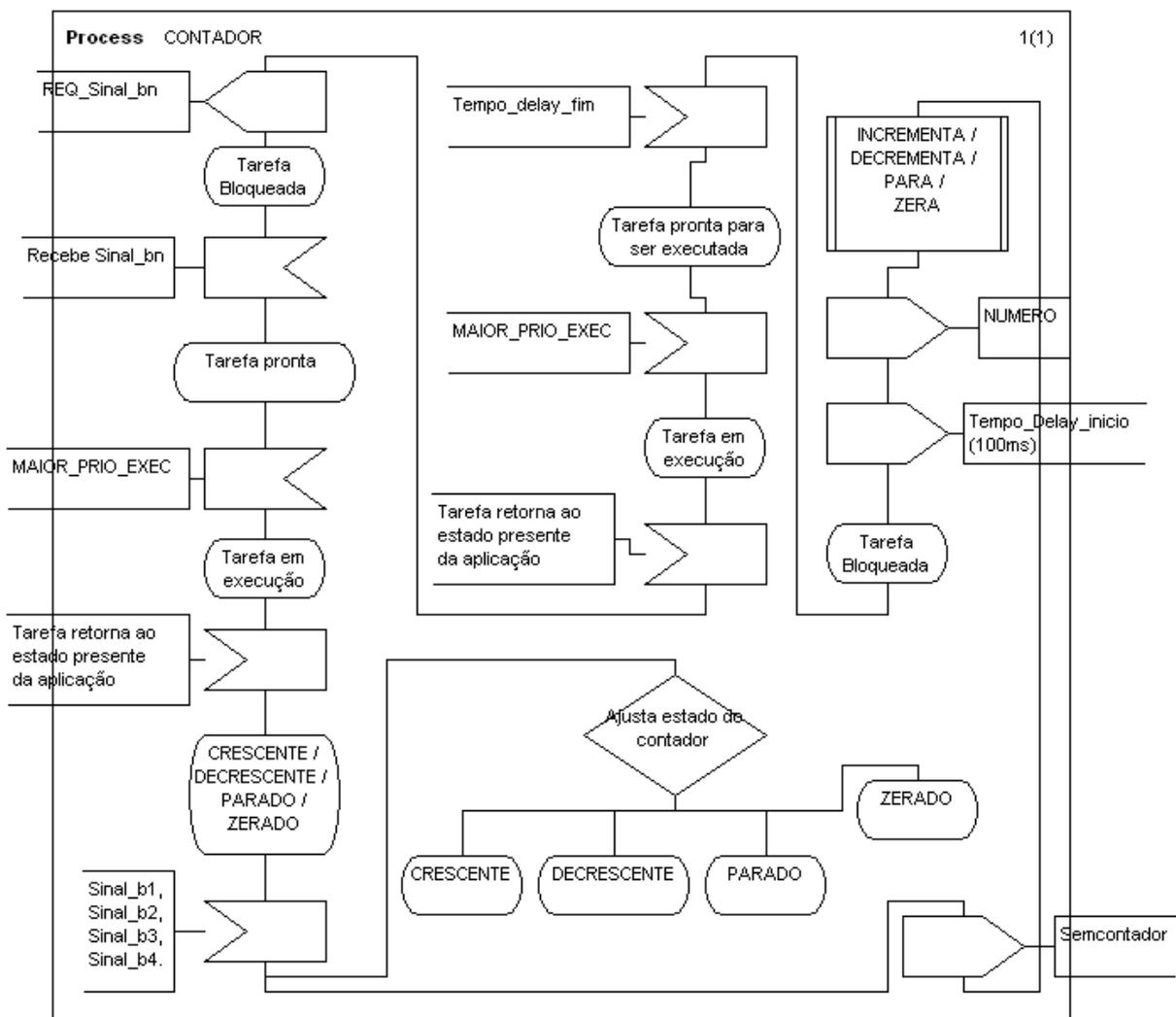



Figura 47. Diagrama SDL do processo CONTADOR com o uso do RTOS.

7.6.3.2.Características da tarefa CONTADOR

- Estados:
  - PARADO;
  - CRESCENTE;
  - DECRESCENTE;
  - ZERO;
  - Tarefa "em execução" – estado determinado pelo kernel;

- o Tarefa "pronta" – estado determinado pelo kernel;
    - o Tarefa "bloqueada" – estado determinado pelo kernel;

- Entradas de dados:
    - o NENHUMA.
- Saídas de dados:
    - o NUMERO (vai para o processo MaqDisplayLEDs) - número de 16 bits que será mostrado no display;
- Sinais de Entrada:
    - o Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4 (vem da tarefa TECLADO). Utiliza o serviço OSMboxPend();
    - o Tempo_Delay_fim (vem do Kernel) – sinal determinado pelo kernel;
    - o MAIOR_PRIO_EXEC (vem do kernel) – sinal determinado pelo kernel.
- Sinais de Saída:
    - o Semcontador (vai para a tarefa TECLADO). Utiliza o serviço OSSemPost;
    - o Tempo_Delay_inicio (vai para o kernel) – sinal determinado pelo kernel. Pertencente ao serviço OSTimeDly();
    - o REQ_Sinal_bn (vai para o kernel) – sinal determinado pelo kernel. Pertencente ao serviço OSMboxPend().

7.6.4.O PROCESSO MaqDisplayLEDs

O processo MaqDisplayLEDs não foi tratado como uma tarefa do sistema e sim como uma rotina chamada na interrupção do TIMER0. Esta escolha deveu-se ao

fato de que quando tratado como tarefa, o processo demonstrou uma cintilação indesejável no display de 7 segmentos, determinada principalmente pela imprecisão da execução de sua tarefa a cada *clock* do sistema (*jitter*), conforme a seção 5.8.2, visto que esta tarefa foi tratada como tarefa de menor prioridade. Como esta rotina deveria ser executada a todo *clock* do relógio, preferiu-se colocá-la na interrupção, o que diminuiu o overhead do sistema e aumentou a precisão do contador. As características deste processo são as mesmas que o da seção 7.5.4 e não serão repetidas aqui.

8.RESULTADOS

8.1.Quantificação das máquinas de estado, estados e sinais dos programas

A Tabela 16 mostra as máquinas de estado necessárias para o desenvolvimento do programa contador crescente/decrescente nos dois casos abordados: utilizando-se um sistema operacional de tempo real (RTOS), e sem o uso deste. Para o programa desenvolvido com o uso do RTOS, foram necessárias 3 máquinas de estado, enquanto que para o programa desenvolvido sem o RTOS, foram necessárias 5 máquinas de estado.

Tabela 16. Número de máquinas de estado utilizadas nos dois programas.

	Contador com RTOS	Contador sem RTOS
Máquinas de estado	TECLADO	TECLADO
	CONTADOR	CONTADOR
	MaqDisplayLEDs	MaqDisplayLEDs
		TIMER_TECLADO
		TIMER_100
Total	3	5

Conforme informa a Tabela 17, para o processo TECLADO foram necessários 2 estados e 4 sinais para o desenvolvimento do programa sem RTOS. Com a utilização do RTOS, o programador não precisou criar nenhum estado e precisou de 3 sinais para desenvolver o programa, 1 a menos do que o programa sem RTOS. Os demais estados (totalizados em 3 estados) e sinais (totalizados em 4 sinais), necessários para o funcionamento correto da máquina de estado TECLADO com RTOS, foram criados e gerenciados pelo kernel a partir dos serviços utilizados pelo programador.

Tabela 17. Número de sinais e estados do processo TECLADO utilizados nos dois programas.

		Contador com RTOS	Kernel	Programador	Contador sem RTOS	Kernel	Programador
TECLADO	Estados	Tarefa "em execução"	x		ESPERA_TECLA		x
		Tarefa "pronta"	x		ESPERA_TIMOUT		x
		Tarefa "bloqueada"	x		-		
	Entradas de dados	-			-		
	Saida de dados	-			-		
	Total de estados		3	0		0	2
	Sinais de entrada	Semcontador		x	Sinal_Timout_Teclado		x
		Tempo_Delay_fim	x		Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4		x
		MAIOR_PRIO_EXEC	x		-		
		Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4		x	-		
	Sinais de saida	Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4		x	Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4		x
		Tempo_Delay_inicio	x		Sinal_Inic_Timout		x
		REQ_Semcontador	x		-		
	Total de sinais		4	3		0	4

Para o desenvolvimento do processo CONTADOR foram necessários 4 estados e 3 sinais no programa sem RTOS; e 7 estados e 7 sinais foram utilizados para o programa com RTOS, conforme mostra a Tabela 18. Destes, 4 estados e 3 sinais foram efetivamente criados pelo programador, sendo o restante criado e gerenciado pelo kernel.

Tabela 18. Número de estados e sinais do processo CONTADOR utilizados nos dois programas.

		Contador com RTOS	Kernel	Programador	Contador sem RTOS	Kernel	Programador
CONTADOR	Estados	PARADO		x	PARADO		x
		CRESCENTE		x	CRESCENTE		x
		DECRESCENTE		x	DECRESCENTE		x
		ZERO		x	ZERO		x
		Tarefa "em execução"	x		-		
		Tarefa "pronta"	x		-		
		Tarefa "bloqueada"	x		-		
	Total de estados		3	4		0	4
	Entradas de dados	-			-		
	Saida de dados	NUMERO		x	NUMERO		x
	Sinais de entrada	Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4		x	Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4		x
		Tempo_Delay_fim	x		Sinal_TIC_100		x
		MAIOR_PRIO_EXEC	x		-		
	Sinais de saida	Semcontador		x	-		
		Tempo_Delay_inicio	x		-		
		REQ_Sinal_bn	x		-		
	Total de sinais		4	3		0	3

O processo MaqDisplayLEDs é idêntico nos dois programas e possui 4 estados e 1 sinal, conforme ilustra a Tabela 19.

Tabela 19. Número de estados e sinais do processo MaqDisplayLEDs utilizados nos dois programas.

		Contador com RTOS	Kernel	Programador	Contador sem RTOS	Kernel	Programador
MaqDisplayLEDs	Estados	DIGITO_3		x	DIGITO_3		x
		DIGITO_2		x	DIGITO_2		x
		DIGITO_1		x	DIGITO_1		x
		DIGITO_0		x	DIGITO_0		x
	Total de estados		0	4		0	4
	Entradas de dados	NUMERO		x	NUMERO		x
	Saida de dados	-			-		
	Sinais de entrada	-			-		
	Sinais de saida	-			-		
	Total de sinais		0	1		0	1

Os processos TIMER_TECLADO e TIMER_100 foram necessários apenas para o desenvolvimento do programa sem o RTOS, não existindo para o programa do contador com RTOS. A Tabela 20 e a Tabela 21 mostram respectivamente, a

quantidade de estados e sinais dos processos TIMER_TECLADO e TIMER_100 para os dois programas.

Tabela 20. Número de sinais e estados do processo TIMER_TECLADO utilizados nos dois programas.

		Contador com RTOS	Kernel	Programador	Contador sem RTOS	Kernel	Programador
TIMER_TECLADO	Estados	-			TIMER_TECL_PARADO		x
		-			TIMER_TECL_CONTANDO		x
	Total de estados						2
	Entradas de dados	-			-		
	Saida de dados	-			-		
	Sinais de entrada	-			Sinal_Inic_Timout		x
	Sinais de saida	-			Sinal_Timout_Teclado		x
	Total de sinais		0	0		0	2

Tabela 21. Número de estados e sinais do processo TIMER_100 utilizados nos dois programas.

		Contador com RTOS	Kernel	Programador	Contador sem RTOS	Kernel	Programador
TIMER_100	Estados	-			Própria rotina		x
	Total de estados		0	0		0	1
	Entradas de dados	-			-		
	Saida de dados	-			-		
	Sinais de entrada	-					
	Sinais de saida	-			Sinal_TIC_100		x
	Total de sinais		0	0		0	1

A quantidade total de máquinas de estado, estados e sinais que o programador precisou criar em cada um dos programas está representada na Tabela 22. A diferença presente nesta tabela reside no fato de que o RTOS possui sinalizadores e serviços que substituem a necessidade de algumas máquinas de estado, e conseqüentemente de seus estados e sinais. No caso dos programas criados para o contador crescente/decrescente, os processos TIMER_TECLADO e TIMER_100 foram totalmente substituídos pelo serviço do kernel OSTimeDly (Figura 48), que foi chamado pelos processos TECLADO e CONTADOR, respectivamente, conforme codificação encontrada no APÊNDICE A.

Tabela 22. Número total de máquinas de estado, estados e sinais criados pelo programador dos dois programas.

Programador	Contador com RTOS	Contador sem RTOS
Máquinas de estado	3	5
Estados	8	13
Sinais	7	11

Como se pode perceber, mesmo numa aplicação simples como o contador crescente/decrescente, o RTOS diminuiu a quantidade de máquinas de estados, estados e sinais que o programador precisou criar para a aplicação. Essa diminuição facilita a codificação do programa na medida em que diminui o número de processos (funções) e variáveis necessárias para o desenvolvimento de uma aplicação.

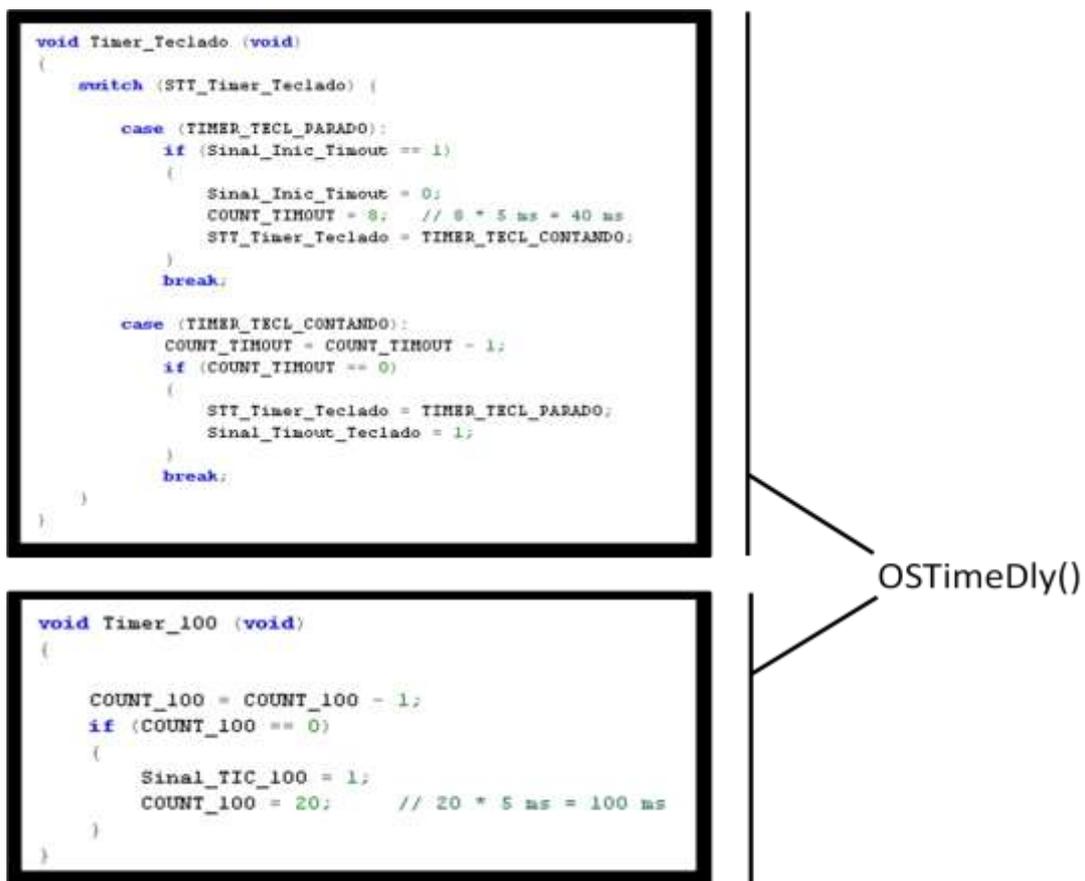



Figura 48. Processos TIMER_TECLADO e TIMER_100 substituídos pelo serviço OSTimeDly()

8.2.Precisão da contagem do contador em cada um dos programas

O requisito de tempo proposto neste trabalho define que o contador crescente/decrescente deve ter uma precisão de 100 ms para cada incremento ou decremento da contagem. Para medir a precisão de cada um dos programas discutidos simulou-se o programa da aplicação no software MPLAB® IDE, através da sua ferramenta de simulação MPLAB SIM. Colocando um breakpoint na linha do código onde ocorre o incremento da contagem nos dois programas aqui discutidos (ver código do processo CONTADOR nos APÊNDICES A E B); e utilizando a ferramenta Stopwatch do MPLAB® IDE, pode-se medir em quanto tempo a variável NUMERO é incrementada. Os resultados obtidos para o programa sem o RTOS e para o programa com o RTOS são apresentas na Figura 49 e na Figura 50, respectivamente.

Conforme a Figura 49, o programa sem o RTOS apresentou uma imprecisão de 773 µs (100,773ms), o que representa 0,77% de erro para cada incremento da contagem. Já o programa com o RTOS apresentou, conforme ilustra a Figura 50, uma imprecisão de 0,072 µs (99,928 ms), ou seja 0,072% de erro, se mostrando, portanto quase 10 vezes mais preciso do que o programa sem o RTOS.

Figura 49. Ilustração da simulação do contador crescente/decrescente sem o uso do RTOS.

Figura 50. Ilustração da simulação do contador crescente/decrescente como uso do RTOS.

9.CONCLUSÃO.

A análise comparativa entre os programas desenvolvidos para o contador crescente/decrescente evidenciou que o uso do sistema operacional de tempo real aumentou a precisão da contagem em quase 10 vezes. Além disso, o RTOS facilitou a codificação do programa ao diminuir o número de funções e variáveis necessárias para o software da aplicação. Funções inteiras como os processos TIMER_TECLADO e TIMER_100 puderam ser completamente substituídos por apenas um serviço de gerenciamento do kernel, no caso do µC/OS-II, pelo serviço OSTimeDly.

Mesmo em uma aplicação simples como o contador crescente/decrescente aqui apresentado, o RTOS trouxe melhorias significativas. Em aplicações complexas que possuem um número elevado de tarefas, estas melhorias são muito mais evidentes e permitem que o programador agregue muito mais confiabilidade ao software desenvolvido com um sistema de tempo real, na medida em que trata com mais precisão os seus requisitos de tempo. Além disso, o RTOS diminui o tempo de desenvolvimento do software por facilitar a codificação do mesmo ao reduzir o tamanho do código e ao dividir o software em módulos quando da criação das tarefas do sistema.

10.REFERÊNCIAS BIBLIOGRÁFICAS.

BITTON, S.; *Fundamentals of Microcontrollers, Sponsored by ARM*. In: techonline Courses & Lectures, 2008. Disponível em <http://techonline.com/learning/course/208800447>. Acesso em: 15 jun. 2009.

BROWN, Nathan; *SputnickOnline.com*, [s. l.], 2005. Disponível em <http://www.sputnickonline.com/projects/programs/micro/uCOS_for_PIC18/index.html>. Acesso em: 10 jun. 2009.

DATA Sheet PIC18FXX2; Microchip. [s.l.]. Microchip Technology Inc, 2006

GANSSLE, Jack et al. Embedded Hardware. Oxford: Elsevier Inc, 2008.

GANSSLE, Jack et al. Embedded Software. Oxford: Elsevier Inc, 2008.

GANSSLE, J.; *Real Time Programming*. [s. l.], 1998. Disponível em <http://www.ganssle.com/articles/realtime.htm>. Acessado em: 15 jun. 2009.

GUIA do usuário Placa de Desenvolvimento MCLab2. [S. l.]: Mosaico Engenharia, 2001. 1 CD-ROM.

LABROSSE, Jean. J; *µC/OS-II, The Real-Time Kernel.* São Francisco: CMP BOOKS, 2002.

LACERDA, W. S.; *Introdução aos Sistemas Embarcados*. In: Palestras de Wilian Soares Lacerda, 2006. Lavras. Disponível em <http://www.dcc.ufla.br/~lacerda/download/palestras/sis_embarcados/palestra_sistemas_embarcados.ppt>. Acesso em: 23 jun. 2009.

LAMIE, William E. Keeping your priorities straight: Part 1 - context switching. [s. l.], 2009. Disponível em < http://www.embedded.com/design/testissue/212902706>. Acessado em: 23 jun. 2009.

LEROUX, Paul N.; SCHAFFER, Jeff. *Exactly When Do You Need Real Time?*. [s. l.], 2006). Disponível em < http://www.embedded.com/columns/technicalinsights/193001454>. Acessado em 29 jun. 2009.

LI, Qing; YAO, Carolyn. *Real-Time Concepts for Embedded Systems*. San Francisco: CMP Books, 2003.

OSHANA, Robert; Real-Time Operating Systems for DSP, part 3. [s. l.], 2007. Disponível em <http://www.dspdesignline.com/199203413;jsessionid=UVCSFO52J3GG4QSNDLOSKH0CJUNN2JVN>. Acessado em: 24 jun, 2009.

STEWART, David B. *Introduction to real time*. [s. l.], 2001. Disponível em <http://www.embedded.com/columns/beginerscorner/9900353> Acessado em: 15 jun.2009.

APÊNDICE A – Código do contador com o uso do RTOS

```
//CONTADOR COM O USO DE RTOS//
/* Arquivo: MAQCONT.c */
/* Versão: 1.0 */
/* Autor: Gabriel Soares Martins - Junho de 2008*/

/* Faculdade de Engenharia Mecânica - Universidade Estadual de Campinas */
#include    "includes.h"
#include    <timers.h>
#include    <delays.h>

// PIC configuration
#pragma config OSC = XT
#pragma config BOR = OFF
#pragma config WDT = OFF
#pragma config LVP = OFF

/*
**==========================================================================
**  1.     DECLARAÇÕES
**  1.1    Constantes Internas
**==========================================================================
*/
// Definição dos estados do processo Contador
#define PARADO                      0
#define CRESCENTE                   1
#define DECRESCENTE         2
#define ZERO                        3

// Definição dos estados do processo Display de LEDs
#define DIGITO_0            0
#define DIGITO_1            1
#define DIGITO_2            2
#define DIGITO_3            3


/*
**==========================================================================
**  1.2    Variáveis globais
**==========================================================================
*/

int                                 NUMERO; // variável necessária para armazenar a contagem do
                                                    // contador e enviá-la para a rotina
atualiza_display,
                                                    // onde o número será "quebrado" em 4 dígitos

unsigned char               NUMERO_BCD[4], //vetor que será utilizado para guardar os 4 dígitos de NUMERO
                                    LED[4], // vetor que será utilizado para guardar os 4 dígitos de
NUMERO
                                    err; //variável utilizada para armazenar código de erro dos serviços do
microC/OS-II

INT8U                               STT_Display_LEDs= DIGITO_3; // variável que guarda o estado do processo
Display de LEDs

        //Esta variável deve ser global para as várias instâncias da

        //função MaqDisplayLEDs manipularem uma variável comum a todas.
/*
**==========================================================================
**  1.3    Protótipos de funções internas (definidas na Seção 5)
**                          (Rotinas construídas para melhorar a modularidade do código)
**==========================================================================
*/
void escreve_display_LEDs( char num_digito, char digito_bcd );
void atualiza_display ( void );

//******************************************************************************
//              Função  MaqDisplayLEDs ()
//******************************************************************************
// Descrição:                   Mostra valor no display de LEDs. Seleciona o dígito que
//                              será aceso e chama a função que escreve no dígito.
```

```
//					A cada tic da base de tempo de 100 ms comuta para o próximo
//					dígito do display. Tem quatro estados,
//
// Estados:		DIGITO_3, DIGITO_2, DIGITO_1, DIGITO_0; ---> um para cada dígito usado no display de LEDs.
//
// Entradas de dados:	NENHUMA
//
// Saidas de dados:	NENHUMA
//
// Sinais de Entrada:	NENHUM
//
// Sinais de Saida:	NENHUM
//
// Chamada por	:	CPUlowInterruptHook
// Chama		:	escreve_display_LEDs
//
// OBSERVAÇÕES :	Opera sobre as variáveis globais LED[0]:LED[2].
//-----------------------------------------------------------------------------
void MaqDisplayLEDs (void)
{
	switch (STT_Display_LEDs) {
			case (DIGITO_3):
				escreve_display_LEDs ( DIGITO_3, LED[0]);
				break;
			case (DIGITO_2):
				escreve_display_LEDs ( DIGITO_2, LED[1]);
				break;
			case (DIGITO_1):
				escreve_display_LEDs ( DIGITO_1, LED[2]);
				break;
			case (DIGITO_0):
				escreve_display_LEDs ( DIGITO_0, LED[3]);
				break;
		}
		STT_Display_LEDs = STT_Display_LEDs + 1;		// Muda estado do processo.
		if ( STT_Display_LEDs > 3) STT_Display_LEDs = 0;
}

/*******************************************************************************
**=============================================================================
**	2				INICIO DA CONFIGURAÇÃO DO RTOS
**=============================================================================
*******************************************************************************/

//**=============================================================================
//** 2.1			Construção da pilha de dados das tarefas
//**=============================================================================

OS_STK  StarttasksStk[100L];
OS_STK  TecladoStk[300L];
OS_STK  ContadorStk[300L];

//**=============================================================================
//** 2.2			Declaração dos eventos a serem utilizados
//**=============================================================================

OS_EVENT *Semcontador; //1 semáforo
OS_EVENT *CORREIO; //1 caixa de correio


//**=============================================================================
//** 2.3						TAREFAS
//**=============================================================================

//*******************************************************************************
//				Função    Teclado ( )
//*******************************************************************************
// Descrição: Lê a tecla e verifica se houve alteração no seu estado.
//	Se houve alteração, espera por um tempo para tratar do debounce. Depois
//	verifica se a tecla	continua pressionada ou não. Caso sim, reconhece que
//				a tecla foi pressioada e envia o sinal correspondente
//				para o processo Contador.
//
// Estados: Tarefa "em execução", Tarefa "pronta", Tarefa "bloqueada"; //estados invisíveis para o programador
//
```

```
// Entradas de dados:             NENHUMA
//
// Saidas de dados  :             NENHUMA
//
// Sinais de Entrada: Semcontador <--- Contador (SEMÁFORO enviado DO CONTADOR
//                                                                 PARA AVISAR QUE ESTE RECEBEU A MENSAGEM)
//
//                                             Tempo_Delay_fim <--- kernel OSTimeDly() //Sinal invísivel para o
programador
//
//                                             MAIOR_PRIO_EXEC <---     kernel escalonador //Sinal invísivel para o
programador
//
//                                             Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4 <--- pressionamento das teclas
//
// Sinais de Saida:   ENVIA_SINAL_CONTADOR (Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4) ---> Contador
//                                            Tempo_Delay_inicio ---> kernel OSTimeDly() //Sinal invísivel para o
programador
//                                            REQ_Semcontador ---> kernel OSSemPend() //Sinal invísivel para o
programador
//
        enviado para requisitar o semáforo para o kernel
//
// Chamada por       :            Escalonador do Kernel
// Chama             :            NINGUÉM
//------------------------------------------------------------------------------

void Teclado (void *pdata){

INT8U    ENVIA_SINAL_CONTADOR=0, //variável que sinalizará para o Contador
                                                                 //qual será o seu estado.
                //********************************************************
                Sinal_s1=0,            // Variáveis que se transformarão no sinal
                Sinal_s2=0,            // enviado para a máquina de estados do contador
                Sinal_s3=0,
                Sinal_s4=0,
                //*********************************************************
                TECLAS=0x00, //variável que guarda o estado atual do TECLADO
                OLD_TECLAS=0x00; //variável que guarda o estado anterior do TECLADO

        pdata=pdata;
        for(;;){
                TECLAS = PORTB & 0x0F;
                if (TECLAS != OLD_TECLAS){
                        OLD_TECLAS=TECLAS;
                        OSTimeDly(8); //requisito de tempo necessário para tratamento do debounce das teclas - 40ms
                        TECLAS = PORTB & 0x0F;
                        if (TECLAS == OLD_TECLAS){
                                Sinal_s1 = TECLAS & 0x01;
                                Sinal_s2 = TECLAS & 0x02;
                                Sinal_s3 = TECLAS & 0x04;
                                Sinal_s4 = TECLAS & 0x08;
                                if ( Sinal_s1 == 0 )
                                        ENVIA_SINAL_CONTADOR = 1; //Sinal_b1
                                if ( Sinal_s2 == 0 )
                                        ENVIA_SINAL_CONTADOR = 2; //Sinal_b2
                                if ( Sinal_s3 == 0 )
                                        ENVIA_SINAL_CONTADOR = 3; //Sinal_b3
                                if ( Sinal_s4 == 0 )
                                        ENVIA_SINAL_CONTADOR = 4; //Sinal_b4
                        }
                        else ENVIA_SINAL_CONTADOR = 0; //Sinal de que não houve alteração
                }
                else ENVIA_SINAL_CONTADOR = 0;    //é necessário enviar o sinal para o contador,
                                                                         //mesmo se não houve
mudança, para desbloquear
                                                                         //a tarefa do contador

                OSMboxPost(CORREIO, &ENVIA_SINAL_CONTADOR);
                OSSemPend(Semcontador, 0, &err); //aguarda resposta do contador
                                                                          //do recebimento da
mensagem enviada
        }
}
```

```
//******************************************************************************
//                                                          Função    Contador ( )
//******************************************************************************
// Descrição:Ou está parado; ou zera contagem; ou faz contagem crescente; ou contagem
//                      decrescente conforme os botões acionados. Tudo a 10Hz ou 100ms.
//
// Estados: PARADO; CRESCENTE; DECRESCENTE; ZERO.
//                      Tarefa "em execução", Tarefa "pronta", Tarefa "bloqueada"; //estados invisíveis para o
programador
//
// Entradas de dados:       NENHUMA
//
// Saidas de dados: variável global NUMERO: número de 16 bits que será mostrado no display
//
// Sinais de Entrada:       ENVIA_SINAL_CONTADOR (Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4) <-- Teclado
//
//                                      Tempo_Delay_fim <--- kernel OSTimeDly() //Sinal invísivel para o
programador
//
//                                      MAIOR_PRIO_EXEC <---     kernel escalonador //Sinal invísivel
para o programador
//
// Sinais de Saida   :      Semcontador --> Teclado
//
//                                      Tempo_Delay_inicio ---> kernel OSTimeDly() //Sinal invísivel para o
programador
//
//                                      REQ_Sinal_bn ---> Kernel OSMboxPend() //Sinal invísivel para o
programador
//
                            //requisita ao kernel receber uma mensagem
//
// Chamada por              : Escalonador do kernel
//
// Chama                    : atualiza_display
//
// OBSERVAÇÕES              : o valor de NUMERO, que é uma variável global, é convertido
//                                  em uma string de caracteres bcd pela rotina atualiza_display.
//------------------------------------------------------------------------------

void Contador (void *pdata)
{
        int CONTAGEM=0; //variável que armazenará a contagem do contador
        INT8U *REC_SINAL_CONTADOR, //variável que receberá o sinal enviado pelo teclado
                STT_Contador=CRESCENTE; // variável que guarda os estados do Contador

        pdata=pdata;
        for(;;){
                REC_SINAL_CONTADOR=(INT8U *)OSMboxPend(CORREIO, 0, &err);
                OSSemPost(Semcontador);
                switch (STT_Contador) {
                        case (PARADO):
                                if (*REC_SINAL_CONTADOR!=0){
                                        if (*REC_SINAL_CONTADOR == 1) //Sinal_b1
                                                STT_Contador = CRESCENTE;
                                        if (*REC_SINAL_CONTADOR == 2) //Sinal_b2
                                                STT_Contador = DECRESCENTE;
                                        if (*REC_SINAL_CONTADOR == 3) //Sinal_b3
                                                STT_Contador = PARADO;
                                        if (*REC_SINAL_CONTADOR == 4) //Sinal_b4
                                                STT_Contador = ZERO;
                                }
                                break;

                        case (CRESCENTE):
                                if (*REC_SINAL_CONTADOR!=0){
                                        if (*REC_SINAL_CONTADOR == 1) //Sinal_b1
                                                STT_Contador = CRESCENTE;
                                        if (*REC_SINAL_CONTADOR == 2) //Sinal_b2
                                                STT_Contador = DECRESCENTE;
                                        if (*REC_SINAL_CONTADOR == 3) //Sinal_b3
                                                STT_Contador = PARADO;
                                        if (*REC_SINAL_CONTADOR == 4) //Sinal_b4
                                                STT_Contador = ZERO;
                                }
```

```
                                        CONTAGEM = CONTAGEM + 1;
                                        if (CONTAGEM > 9999) CONTAGEM = 0;
                                        NUMERO = CONTAGEM;
                                        atualiza_display ();
                                        break;

                                case (DECRESCENTE):
                                        if (*REC_SINAL_CONTADOR!=0){
                                                if (*REC_SINAL_CONTADOR == 1) //Sinal_b1
                                                        STT_Contador = CRESCENTE;
                                                if (*REC_SINAL_CONTADOR == 2) //Sinal_b2
                                                        STT_Contador = DECRESCENTE;
                                                if (*REC_SINAL_CONTADOR == 3) //Sinal_b3
                                                        STT_Contador = PARADO;
                                                if (*REC_SINAL_CONTADOR == 4) //Sinal_b4
                                                        STT_Contador = ZERO;
                                        }
                                        CONTAGEM = CONTAGEM - 1;
                                        if (CONTAGEM < 0) CONTAGEM = 9999;
                                        NUMERO = CONTAGEM;
                                        atualiza_display ();
                                        break;

                                case (ZERO):
                                        if (*REC_SINAL_CONTADOR!=0){
                                                if (*REC_SINAL_CONTADOR == 1) //Sinal_b1
                                                        STT_Contador = CRESCENTE;
                                                if (*REC_SINAL_CONTADOR == 2) //Sinal_b2
                                                        STT_Contador = DECRESCENTE;
                                                if (*REC_SINAL_CONTADOR == 3) //Sinal_b3
                                                        STT_Contador = PARADO;
                                                if (*REC_SINAL_CONTADOR == 4) //Sinal_b4
                                                        STT_Contador = ZERO;
                                        }
                                        CONTAGEM = 0;
                                        NUMERO = CONTAGEM;
                                        atualiza_display ();
                                        break;
                        }
                        OSTimeDly(20);      //requisito de tempo necessário para unidade da contagem ser 100ms
                }
}

//************************************************************************************
//                                                                               Função Starttasks()
//************************************************************************************

// Descrição: Função que inicializa toda aplicação
//------------------------------------------------------------------------------------

void Starttasks(void *pdata){

   #if OS_CRITICAL_METHOD ==3
      OS_CPU_SR cpu_sr;
   #endif

   pdata = pdata;

//**==========================================================================
//**                        Configurando os pinos de entrada e saída (I/O´s)
//**==========================================================================

   ADCON1  = 0x07;      // Configura registrador A/D --> Esta configuração determina que as entradas são digitais

   PORTA = 0x00;
   TRISA = 0x00;

   PORTB = 0xF0;        // Os pinos RB0 a RB3 são as
          TRISB = 0x0F;                        // entradas das teclas
                                                          // RB4 a RB7 sao seletores dos
                                                          // displays de 7 segmentos e devem ser saídas.

   PORTC = 0x00;
   TRISC = 0x00;
   PORTD = 0x00;
```

```
    TRISD = 0x00;
    PORTE = 0x00;
    TRISE = 0x00;

//**==========================================================================
//**                    Configurando o relógio de Tempo real
//**==========================================================================

    OS_ENTER_CRITICAL();
    OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_1);
    WriteTimer0(-5000); //tick do relógio do sistema deve ter resolução de 5ms--refinamento.
    OS_EXIT_CRITICAL();


//**==========================================================================
//**                    Inicializando as variáveis globais do sistema
//**==========================================================================

    LED[0]                                  = 0;
            LED[1]                          = 0;
            LED[2]                          = 0;
            LED[3]                          = 0;
            NUMERO_BCD[0]                   = 0;
            NUMERO_BCD[1]                   = 0;
            NUMERO_BCD[2]                   = 0;
            NUMERO_BCD[3]                   = 0;
            NUMERO_BCD[4]                   = 0;
            NUMERO                          = 0;


//**==========================================================================
//**                                  Criando os eventos
//**==========================================================================

            Semcontador=OSSemCreate(0);
    CORREIO=OSMboxCreate(NULL);


//**==========================================================================
//**                                  Criando as tarefas do usuário
//**==========================================================================

    OSTaskCreate(Teclado, NULL, &TecladoStk[0], 3);
    OSTaskCreate(Contador, NULL, &ContadorStk[0], 1);

//**==========================================================================
//**                          Deletando a tarefa de inicialização do sistema
//**==========================================================================

    OSTaskDel(OS_PRIO_SELF);
}

//**==========================================================================
//** 2.4                      Início da Função Principal
//**==========================================================================
void main (void)
{
    INTCONbits.GIEH = 0;      // Habilitando as interrupções do sistema
    OSInit(); //Inicializando o RTOS
    OSTaskCreate(Starttasks, NULL, &StarttasksStk[0], 0); //Criando a tarefa que inicializará a aplicação
    OSStart(); //Iniciando o RTOS e transferindo o controle da CPU para a aplicação.
}
//============================================================================
//                                  FIM DA FUNÇÃO PRINCIPAL
//============================================================================


//**==========================================================================
//** 2.5                      FUNÇÕES AUXILIARES
//**==========================================================================

//*****************************************************************************
//      Função    escreve_display_LEDs( num_digito, digito_bcd )
//*****************************************************************************
// Descrição          :       Escreve valor em um dígito do display de LEDs.
```

```
//
// Entradas de dados:          num_digito - número do dígito em que escreverá (0,1,2,3)
//                                                  digito_bcd - número a ser escrito no display (em BCD) <-- variável
global LED[]

// Saidas de dados  :          NENHUMA
//
// Sinais de Entrada:          NENHUM
// Sinais de Saida   :         NENHUM
//
// Chamada por                 :   Função  MaqDisplayLEDs
// Chama                       :   NINGUÉM
//
// OBSERVAÇÕES:
// Os dígitos são selecionados pelos seus pinos correspondentes do PORTB
//          RB4 = dígito menos significativo
//          RB5
//          RB6
//          RB7 = dígito mais significativo
//
//---------------------------------------------------------------------------
void escreve_display_LEDs( char num_digito, char digito_bcd )
{
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// * TABELA PARA OS DISPLAYS DE 7 SEGMENTOS *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        const char TABELA_7SEG[16]= {
        0x3F,     // 0h - 0
        0x06,     // 1h - 1
        0x5B,     // 2h - 2
        0x4F,     // 3h - 3
        0x66,     // 4h - 4
        0x6D,     // 5h - 5
        0x7D,     // 6h - 6
        0x07,     // 7h - 7
        0x7F,     // 8h - 8
        0x6F,     // 9h - 9
        0x00,     // Ah - A
        0x00,     // Bh - b
        0x00,     // Ch - C
        0x00,     // Dh - d
        0x00,     // Eh - E
        0x00      // Fh - F
        };
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        PORTBbits.RB4 = 0;
        PORTBbits.RB5 = 0;
        PORTBbits.RB6 = 0;
        PORTBbits.RB7 = 0;

        switch (num_digito) {
                case 0:
                        PORTBbits.RB4 = 1;
                        break;
                case 1:
                        PORTBbits.RB5 = 1;
                        break;
                case 2:
                        PORTBbits.RB6 = 1;
                        break;
                case 3:
                        PORTBbits.RB7 = 1;
                        break;
        }

        digito_bcd = digito_bcd & 0x0F;          // Mascara 4 LSb
        PORTD = TABELA_7SEG[digito_bcd];  // Mostra dígito
}


//*****************************************************************************
//      Função    atualiza_display ( CONTAGEM )
//*****************************************************************************
// Descrição:          separa cada dígito numa string
```

```
//                          que será usada pela rotina que aciona o display de LEDs.
//
// Entradas de dados:       variável global NUMERO
// Saidas de dados  :       variável global LED[]
//
// Sinais de Entrada:       NENHUM
//
// Sinais de Saida   :      NENHUM
//
// Chamada por              :       Contador
// Chama                    :       NINGUÉM
//
// OBSERVAÇÕES:             A conversão usa o método das divisões sucessivas.
//------------------------------------------------------------------------------

void atualiza_display ( void )
{
        unsigned char       i,
                                        y;
        int                         x;

        x = NUMERO;
        NUMERO_BCD[4] = 0;                  // End of string
        LED[0] = 0;
        LED[1] = 0;
        LED[2] = 0;
        LED[3] = 0;


        i = 3;
        if (x != 0 ) {
                while (x != 0) {
                        NUMERO_BCD[i] = x % 10;
                        y = NUMERO_BCD[i];
                        LED[i] = y;
                        x = x / 10;
                        i -= 1;
                }
        }
        else {
                NUMERO_BCD[3] = '0';
                NUMERO_BCD[2] = '0';
                NUMERO_BCD[1] = '0';
                NUMERO_BCD[0] = '0';
        }
}
```

APÊNDICE B – Código do contador sem o uso do RTOS

```
//CONTADOR SEM O USO DE RTOS//
/* Arquivo: ProgContadorSDL.c */
/* Versão: 1.0 */
/* Autor: Gabriel Soares Martins - Junho de 2008*/

/* Faculdade de Engenharia Mecânica - Universidade Estadual de Campinas */

#include <p18cxxx.h>      /* Definição das funções intrínsecas do compilador      */
#include <timers.h>

#pragma config WDT = OFF
#pragma config OSC = XT

// Definição dos estados do processo Contador
#define PARADO              0
#define CRESCENTE           1
#define DECRESCENTE 2
#define ZERO                3

// Definição dos estados do processo Timer do Teclado
#define TIMER_TECL_PARADO          0
#define TIMER_TECL_CONTANDO        1

// Definição dos estados do processo Teclado
#define ESPERA_TECLA               0
#define ESPERA_TIMOUT              1

// Definição dos estados do processo Display de LEDs
#define DIGITO_0                   0
#define DIGITO_1                   1
#define DIGITO_2                   2
#define DIGITO_3                   3

//-------------------------------------------------------------------------------------------------
unsigned char       COUNT_100,                  //Contador de 100ms                 //CONTADORES
                    COUNT_TIMOUT,       //Contador do timer do teclado -> 40ms
//-------------------------------------------------------------------------------------------------
                    Sinal_TIC_100,              //Sinal para o contador de 100ms        //SINALIZADORES
                    Sinal_ESTADO_CONTADOR,  //sinal que indicará o estado do contador
                  //Sinal_b1
                            //Sinal_b2
                                                    //Sinal_b3
                                                    //Sinal_b4
                    Sinal_Inic_Timout,      // Sinal de fim da temporização do teclado
              Sinal_Timout_Teclado,         // Sinal de início da temporização do teclado

                    Sinal_s1,  //sinais recebidos pelo PORTB pelo teclado
                    Sinal_s2,
                    Sinal_s3,
                    Sinal_s4,
//-------------------------------------------------------------------------------------------------
                    STT_Teclado,                // Estado do processo Teclado  //ARMAZENADORES DE ESTADO
                    STT_Timer_Teclado,          // Estado do processo Timer do Teclado
                    STT_Display_LEDs,               // Estado do processo Display de LEDs
                    STT_Contador,               // Estado do processo Contador
//-------------------------------------------------------------------------------------------------
                    TECLAS,                     // Variável que armazenará o estado atual das teclas   //variáveis
diversas
                    OLD_TECLAS,                         // Variável que armazenará estado anterior das teclas

              NUMERO_BCD[4], //Vetor que armazenará os digitos do display de 7 segmentos
                    LED[4], //Vetor que armazenará os digitos do display de 7 segmentos

                    i,
                    x,
                    y;

int                 NUMERO,                             // Conteudo do contador
                    CONTAGEM;                           // Conteudo do contador


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
// * TABELA PARA OS DISPLAYS DE 7 SEGMENTOS *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
const char                          TABELA_7SEG[16]= {
                                                        0x3F,       // 0h - 0
                                                        0x06,       // 1h - 1
                                                        0x5B,       // 2h - 2
                                                        0x4F,       // 3h - 3
                                                        0x66,       // 4h - 4
                                                        0x6D,       // 5h - 5
                                                        0x7D,       // 6h - 6
                                                        0x07,       // 7h - 7
                                                        0x7F,       // 8h - 8
                                                        0x6F,       // 9h - 9
                                                        0x00,       // Ah - A
                                                        0x00,       // Bh - b
                                                        0x00,       // Ch - C
                                                        0x00,       // Dh - d
                                                        0x00,       // Eh - E
                                                        0x00        // Fh - F
                                              };
//-----------------------------------------------------------------------------------------------

// Protótipos das funções usadas na rotina de interrupção.
void high_isr(void);
void escreve_display_LEDs( char num_digito, char digito_bcd );
void MaqDisplayLEDs (void);
void Contador (void);
void Teclado (void);
void Timer_Teclado (void);
void atualiza_display (void);
void Timer_100 (void);

#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
_asm GOTO high_isr _endasm
}
#pragma code /* return to the default code section */


/************************************************************************************
 INÍCIO DA FUNÇÃO PRINCIPAL
************************************************************************************/

void main( void)
{
   /* NOTA: dispositivos que têm conversor AD podem precisar
      configurar os pinos assinalados a 'IN' e'OUT' como
      ES digital */
   ADCON1 = 0x07;

   /* Primeiramente decida o nível inicial dos pinos das
      portas de saida, e depois defina a configuração dos
      pinos como entrada/saida. Isto evitará ruidos nos pinos
      de saida. */

   PORTA = 0x00;
   TRISA = 0x00;
   PORTB = 0xF0;
   TRISB = 0x0F;                         // Os pinos RB0 a RB3 são as
                                                                      // entradas das teclas
                                                                      // RB4 a RB7 sao seletores dos
                                                                      // displays de 7 segmentos.
   PORTC = 0x00;
   TRISC = 0x00;
   PORTD = 0x00;
   TRISD = 0x00;
   PORTE = 0x00;
   TRISE = 0x00;

// Inicializa o TIMER0
            OpenTimer0( TIMER_INT_ON            &
            T0_8BIT                                       &
            T0_SOURCE_INT                                 &
            T0_PS_1_128 );
```

```
            WriteTimer0( 256-38 );                    // Atribui valor inicial do TIMER0 tal que
                                                                  // o atrazo total seja 128 us * 39 = 5
ms
                                                                  // para um clock de 4 MHz


/*-------------- INICIALIZAÇÃO DOS PROCESSOS ---------------------------------------*/
CONTAGEM                                    = 0;
COUNT_100                                   = 20;
Sinal_TIC_100                      = 0;
STT_Teclado                                 = ESPERA_TECLA;
STT_Timer_Teclado                  = TIMER_TECL_PARADO;
STT_Display_LEDs           = DIGITO_3;
STT_Contador                       = CRESCENTE;
TECLAS                                      = 0x00;
OLD_TECLAS                                  = 0x00;
Sinal_Inic_Timout          = 1;
Sinal_Timout_Teclado       = 0;
Sinal_s1                           = 1;
Sinal_s2                           = 1;
Sinal_s3                           = 1;
Sinal_s4                           = 1;
COUNT_TIMOUT                       = 0;
NUMERO                                               = 0;
NUMERO_BCD[0]                      = 0;
NUMERO_BCD[1]                      = 0;
NUMERO_BCD[2]                      = 0;
NUMERO_BCD[3]                      = 0;
NUMERO_BCD[4]                      = 0;
LED[0]                                      = 0;
LED[1]                                      = 0;
LED[2]                                      = 0;
LED[3]                                      = 0;
LED[4]                                      = 0;
i                                                    = 0;
x                                                    = 0;
y                                                    = 0;


// Inicializa as interrupções
   INTCONbits.GIE = 1;             // Habilitação global de interrupções


/*---------------------- Laço Principal infinito ------------------------------------*/
   while (1) {}
}

/****************************************************************************************
 FIM DA FUNÇÃO PRINCIPAL
****************************************************************************************/


//*******************************************************************************
/* ROTINA DE SERVIÇO DA INTERRUPCÃO DO TIMER0
            Esta rotina distribui o tempo do processador entre os processos
            Como todos os processos são chamados a cada interrupção, esta
            rotina é chamada de Executor Cíclico */
//--------------------------------------------------------------------------

#pragma interrupt high_isr
void high_isr (void)
{   if (INTCONbits.TMR0IF)               // Se o TIMER0 gerou um pedido de interrupção
            {
// ----------------------------------------------- Rotina de serviço da interrupção do TIMER0
                                                                  // TIMER0 interrompe a cada 5 ms
      WriteTimer0( 256-38) ;   // 128 us * 39 = 5 ms @ 4MHz
                                                                  // Prescaler = 128
      INTCONbits.TMR0IF = 0;                // Zera flag de interrupção do TIMER0
//*************************************************************************************************************
// Aqui são chamados os processos do sistema, isto é, as máquinas de estado definidas
// no diagrama de partição do sistema e detalhadas com os diagramas SDL.
                        MaqDisplayLEDs ();          // Processo que aciona o display de LEDs
                        Contador ();                // Processo que implementa o contador crescente/decrescente
                        Teclado ();                 // Processo que faz a leitura e "debounce" do teclado
```

```
                        Timer_Teclado ();    // Processo do timer do teclado
                        Timer_100 ();                        // Processo da base de tempo de 4 Hz do contador
   }
}
//****************************************************************************************************




//*******************************************************************************************
//*******************************************************************************************
//*******************************************************************************************
//                                                                                          M Á Q U I N A S   D E   E S T A D O
S
//*******************************************************************************************
//*******************************************************************************************
//*******************************************************************************************

//*************************************************************************
//                  Função  MaqDisplayLEDs ()
//
// Descrição:                       Mostra valor no display de LEDs. Seleciona o dígito que
//                                  será aceso e chama a função que escreve no dígito.
//                                  A cada tic da base de tempo de 100 ms comuta para o próximo
//                                  dígito do display. Tem 4 estados, um para cada dígito
//                                  usado no display de LEDs.
//
// Estados: DIGITO_3; DIGITO_2; DIGITO_1; DIGITO_0;
//
// Entradas de dados:               LED[] <--- atualiza_display () //Variavel NUMERO separada em cada dígito do display
//
// Saidas de dados:                 NENHUMA
//
// Sinais de Entrada:               NENHUM
//
// Sinais de Saida:                 NENHUM
//
// Chamada por      :               int_server ----> interrupção void high_isr ()
//
// Chama            :               escreve_display_LEDs ()
//
//---------------------------------------------------------------------------
void MaqDisplayLEDs (void)
{
            switch (STT_Display_LEDs) {
                        case (DIGITO_3):
                                    escreve_display_LEDs ( DIGITO_3, LED[0]);
                                    break;
                        case (DIGITO_2):
                                    escreve_display_LEDs ( DIGITO_2, LED[1]);
                                    break;
                        case (DIGITO_1):
                                    escreve_display_LEDs ( DIGITO_1, LED[2]);
                                    break;
                        case (DIGITO_0):
                                    escreve_display_LEDs ( DIGITO_0, LED[3]);
                                    break;
            }
            STT_Display_LEDs = STT_Display_LEDs + 1;              // Muda estado do processo.
            if ( STT_Display_LEDs > 3) STT_Display_LEDs = 0;
}

//*************************************************************************
//          Função    Contador ( )
//
// Descrição:           Ou está parado; ou faz contagem crescente; ou contagem
//                                  decrescente; ou zera contagem, conforme os botões acionados.
//                                  Tudo a 10 Hz ou 100ms.
//
// Estados: PARADO; CRESCENTE; DECRESCENTE; ZERO.
//
// Entradas de dados:               NENHUMA
//
// Saidas de dados: NUMERO --> atualiza_display () //número de 16 bits que será mostrado no display
//
```

```
// Sinais de Entrada:              Sinal_ESTADO_CONTADOR       <--       Teclado () //Sinal_b1, Sinal_b2, Sinal_b3,
Sinal_b4
//                                              Sinal_TIC_100            <--              Timer_100 ()
//
// Sinais de Saida    :                  NENHUM
//
// Chamada por                     : int_server //interrupção void high_isr ()
//
// Chama                           : atualiza_display ()
//
// OBSERVAÇÕES                     : o valor de NUMERO, que é uma variável global, é convertido
//                                          em uma string de caracteres bcd pela rotina atualiza_display.
//---------------------------------------------------------------------------

void Contador (void)
{
        switch (STT_Contador) {
                case (PARADO):
                        if (Sinal_ESTADO_CONTADOR != 0){
                                if (Sinal_ESTADO_CONTADOR == 1) //Sinal_b1
                                {
                                        Sinal_ESTADO_CONTADOR = 0; // Um sinal deve sempre ser "consumido"
                                        STT_Contador = CRESCENTE;
                                }
                                if (Sinal_ESTADO_CONTADOR == 2) //Sinal_b2
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = DECRESCENTE;
                                }
                                if (Sinal_ESTADO_CONTADOR == 3) //Sinal_b3
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = PARADO;
                                }
                                if (Sinal_ESTADO_CONTADOR == 4) //Sinal_b4
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = ZERO;
                                }
                        }
                        if (Sinal_TIC_100 == 1)
                        {
                                Sinal_TIC_100 = 0;
                        }
                        break;

                case (CRESCENTE):
                        if (Sinal_ESTADO_CONTADOR != 0){
                                if (Sinal_ESTADO_CONTADOR == 1) //Sinal_b1
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = CRESCENTE;
                                }
                                if (Sinal_ESTADO_CONTADOR == 2) //Sinal_b2
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = DECRESCENTE;
                                }
                                if (Sinal_ESTADO_CONTADOR == 3) //Sinal_b3
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = PARADO;
                                }
                                if (Sinal_ESTADO_CONTADOR == 4) //Sinal_b4
                                {
                                        Sinal_ESTADO_CONTADOR = 0;
                                        STT_Contador = ZERO;
                                }
                        }
                        if (Sinal_TIC_100 == 1)
                        {
                                Sinal_TIC_100 = 0;
                                CONTAGEM = CONTAGEM + 1;
                                if (CONTAGEM > 9999) CONTAGEM = 0;
                                NUMERO = CONTAGEM;
```

```
                        atualiza_display ();
                }
                break;

        case (DECRESCENTE):
                if (Sinal_ESTADO_CONTADOR != 0){
                        if (Sinal_ESTADO_CONTADOR == 1) //Sinal_b1
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = CRESCENTE;
                        }
                        if (Sinal_ESTADO_CONTADOR == 2) //Sinal_b2
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = DECRESCENTE;
                        }
                        if (Sinal_ESTADO_CONTADOR == 3) //Sinal_b3
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = PARADO;
                        }
                        if (Sinal_ESTADO_CONTADOR == 4) //Sinal_b4
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = ZERO;
                        }
                }
                if (Sinal_TIC_100 == 1)
                {
                        Sinal_TIC_100 = 0;
                        CONTAGEM = CONTAGEM - 1;
                        if (CONTAGEM < 0) CONTAGEM = 9999;
                        NUMERO = CONTAGEM;
                        atualiza_display ();
                }
                break;

        case (ZERO):
                if (Sinal_ESTADO_CONTADOR != 0){
                        if (Sinal_ESTADO_CONTADOR == 1) //Sinal_b1
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = CRESCENTE;
                        }
                        if (Sinal_ESTADO_CONTADOR == 2) //Sinal_b2
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = DECRESCENTE;
                        }
                        if (Sinal_ESTADO_CONTADOR == 3) //Sinal_b3
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = PARADO;
                        }
                        if (Sinal_ESTADO_CONTADOR == 4) //Sinal_b4
                        {
                                Sinal_ESTADO_CONTADOR = 0;
                                STT_Contador = ZERO;
                        }
                }
                if (Sinal_TIC_100 == 1)
                {
                        Sinal_TIC_100 = 0;
                        CONTAGEM = 0;
                        NUMERO = CONTAGEM;
                        atualiza_display ();
                }
                break;
    }
}


//*******************************************************************************
//      Função   Teclado ( )
// Descrição:           É uma máquina de dois estados. ESPERA_TECLA
```

```
//                                  e ESPERA_TIMOUT. No primeiro, aguarda que seja detectado
//                                  que uma tecla foi pressionada, quando dispara o timer
//                                  de debounce do teclado e fica aguardando o TIMEOUT.
//                                  Quando chega o sinal de TIMEOUT, verifica se a tecla
//                                  continua pressionada ou não. Caso sim, reconhece que
//                                  a tecla foi pressionada e envia o sinal correspondente
//                                  para o processo Contador. Volta ao estado inicial.
//
// Estados: ESPERA_TECLA; ESPERA_TIMOUT;
//
// Entradas de dados:               NENHUMA
//
// Saidas de dados  :               NENHUMA
//
// Sinais de Entrada:               Sinal_Timout_Teclado <-- Timer_teclado ()
//                                                      Sinal_s1, Sinal_s2, Sinal_s3, Sinal_s4 <--      sinais enviados
pelo teclado
//
// Sinais de Saida:   Sinal_ESTADO_CONTADOR  -->        Contador () //Sinal_b1, Sinal_b2, Sinal_b3, Sinal_b4
//                                              Sinal_Inic_Timout --> Timer_teclado ()
//
// Chamada por      :               int_server //interrupção void high_isr ()
// Chama            :               NINGUÉM
//
// OBSERVAÇÕES :                    esta máquina de estados recebe sinais tanto do hardware
//                                              (teclas via PORTB) quanto de outra máquina
//                                              implementada por software: Timer_teclado.
//------------------------------------------------------------------------------

void Teclado (void)
{
        switch (STT_Teclado) {

                case (ESPERA_TECLA):
                        TECLAS = PORTB & 0x0F;
                        if (TECLAS != OLD_TECLAS)
                        {
                                OLD_TECLAS = TECLAS;
                                Sinal_Inic_Timout = 1;
                                STT_Teclado = ESPERA_TIMOUT;
                        }
                        break;

                case (ESPERA_TIMOUT):
                        if (Sinal_Timout_Teclado == 1)
                        {
                                Sinal_Timout_Teclado = 0;
                                STT_Teclado = ESPERA_TECLA;
                                TECLAS = PORTB & 0x0F;
                                if (TECLAS == OLD_TECLAS)
                                {
                                        Sinal_s1 = TECLAS & 0x01;
                                        Sinal_s2 = TECLAS & 0x02;
                                        Sinal_s3 = TECLAS & 0x04;
                                        Sinal_s4 = TECLAS & 0x08;

                                        if ( Sinal_s1 == 0 )
                                                Sinal_ESTADO_CONTADOR = 1; //Sinal_b1
                                        if ( Sinal_s2 == 0 )
                                                Sinal_ESTADO_CONTADOR = 2; //Sinal_b2
                                        if ( Sinal_s3 == 0 )
                                                Sinal_ESTADO_CONTADOR = 3; //Sinal_b3
                                        if ( Sinal_s4 == 0 )
                                                Sinal_ESTADO_CONTADOR = 4; //Sinal_b4
                                }
                        }
                        break;
        }
}


//******************************************************************************
//      Função    Timer_teclado ( )
//
```

```
// Descrição:                     Esta máquina de estados realiza a  temporização de 40 ms
//                                da eliminação de repique do teclado (debounce). Tem dois
//                                estados: TIMER_TECL_PARADO e TIMER_TECL_CONTANDO. No pri-
//                                meiro estado, aguarda o sinal para iniciar a temporização,
//                                e no segundo estado decrementa o contador até que este
//                                chegue a zero, quando emite o sinal de TIMOUT para o
//                                processo do Teclado e volta para o estado inicial.
//
// Estados: TIMER_TECL_PARADO e TIMER_TECL_CONTANDO
//
// Entradas de dados:             NENHUMA
// Saidas de dados  :             NENHUMA
//
// Sinais de Entrada:             Sinal_Inic_Timout   <--         Teclado ()
//
// Sinais de Saida   :            Sinal_Timout_Teclado --> Teclado ()
//
// Chamada por                    :          int_server ----> interrupção void high_isr ()
// Chama                          :          NINGUÉM
//
// OBSERVAÇÕES                    :          Os timers ou temporizadores executam a função de medir
//                                                      um intervalo de tempo relativo à sua ativação.
//---------------------------------------------------------------------------

void Timer_Teclado (void)
{
        switch (STT_Timer_Teclado) {

                case (TIMER_TECL_PARADO):
                        if (Sinal_Inic_Timout == 1)
                        {
                                Sinal_Inic_Timout = 0;
                                COUNT_TIMOUT = 8;          // 8 * 5 ms = 40 ms
                                STT_Timer_Teclado = TIMER_TECL_CONTANDO;
                        }
                        break;

                case (TIMER_TECL_CONTANDO):
                        COUNT_TIMOUT = COUNT_TIMOUT - 1;
                        if (COUNT_TIMOUT == 0)
                        {
                                STT_Timer_Teclado = TIMER_TECL_PARADO;
                                Sinal_Timout_Teclado = 1;
                        }
                        break;
        }
}

//******************************************************************************
//        Função    Timer_100 ( )
//
// Descrição: Temporiza o intervalo de 100 milisegundos entre incrementos ou decrementos
// do contador.
//
// Estados: NENHUM;
//
// Entradas de dados:             NENHUMA
// Saida de dados    :            NENHUMA
//
// Sinais de Entrada:             NENHUM
//
// Sinais de Saida   :            Sinal_TIC_100 ---> Contador ()
//
// Chamada por                    :          int_server
//
// Chama                          :          NINGUÉM
//
// OBSERVAÇÕES                    :          nenhuma
//---------------------------------------------------------------------------

void Timer_100 (void)
{

        COUNT_100 = COUNT_100 - 1;
        if (COUNT_100 == 0)
```

```
        {
                Sinal_TIC_100 = 1;
                COUNT_100 = 20;             // 20 * 5 ms = 100 ms
        }
}
//-----------------------------------------------------------------------------------------------------------
//***********************************************************************************************************
//                                                                                         FUNÇÕES AUXILIARES
//***********************************************************************************************************
//-----------------------------------------------------------------------------------------------------------
//******************************************************************************
//      Função   atualiza_display ( CONTAGEM )
//
// Descrição:                   Separa cada dígito numa string
//                              que será usada pela rotina que aciona o display de LEDs.
//
// Entradas de dados:           NUMERO <--- Contador ()
// Saidas de dados  :           LED[] ---> MaqDisplayLEDs ()
//
// Sinais de Entrada:           NENHUM
//
// Sinais de Saida   :          NENHUM
//
// Chamada por                  :       Contador ()
// Chama                        :       NINGUÉM
//
// OBSERVAÇÕES:                 A conversão usa o método das divisões sucessivas.
//------------------------------------------------------------------------------

void atualiza_display ( void )
{
        unsigned char i, y;
        int x;
        x = NUMERO;
        NUMERO_BCD[4] = 0;                  // End of string
        LED[0] = 0;
        LED[1] = 0;
        LED[2] = 0;
        LED[3] = 0;


        i = 3;
        if (x != 0 ) {
                while (x != 0) {
                        NUMERO_BCD[i] = x % 10;
                        y = NUMERO_BCD[i];
                        LED[i] = y;
                        x = x / 10;
                        i -= 1;
                }
        }
        else {
                NUMERO_BCD[3] = '0';
                NUMERO_BCD[2] = '0';
                NUMERO_BCD[1] = '0';
                NUMERO_BCD[0] = '0';
        }
}

//******************************************************************************
//      Função   escreve_display_LEDs( num_digito, digito_bcd )
//
// Descrição                    :       Escreve valor em um dígito do display de LEDs.
//
// Entradas de dados:           número do dígito em que escreverá (0,1,2,3)
//                                          número a ser escrito no display (em BCD) <-- variável global LED[]
//
// Saidas de dados  :           NENHUMA
//
// Sinais de Entrada:           NENHUM
//
// Sinais de Saida   :          NENHUM
//
// Chamada por                  :  MaqDisplayLEDs ()
//
```

```
// Chama                    :  NINGUÉM
//
// OBSERVAÇÕES:
//                                          Os dígitos são selecionados pelos seus pinos correspondentes do PORTB
//                                          RB4 = dígito menos significativo
//                                          RB5
//                                          RB6
//                                          RB7 = dígito mais significativo
//
//-----------------------------------------------------------------------------
void escreve_display_LEDs( char num_digito, char digito_bcd )
{
        PORTBbits.RB4 = 0;
        PORTBbits.RB5 = 0;
        PORTBbits.RB6 = 0;
        PORTBbits.RB7 = 0;

        switch (num_digito) {
                case 0:
                        PORTBbits.RB4 = 1;
                        break;
                case 1:
                        PORTBbits.RB5 = 1;
                        break;
                case 2:
                        PORTBbits.RB6 = 1;
                        break;
                case 3:
                        PORTBbits.RB7 = 1;
                        break;
        }

        digito_bcd = digito_bcd & 0x0F;         // Mascara 4 LSb
        PORTD = TABELA_7SEG[digito_bcd];   // Mostra dígito
}
```

APÊNDICE C - Manual de portabilidade do Micrium µC/OS-II v2.86 para o Microchip PIC18F452.

Nathan Brown codificou os principais arquivos do µC/OS-II para portá-lo na família PIC18F. Estes arquivos, com o código específico do microcontrolador PIC18F podem ser encontrados no endereço:

- *http://www.sputnickonline.com/projects/programs/micro/uCOS_for_PIC18/files/MPLAB-C18v101.zip.*

No entanto, este porte é válido para a versão 2.51 do Micrium µC/OS-II, utilizando como compilador o Microchip MPLAB C18 compiler na sua versão 2.09.24 até a 3.01. David Fischer, a partir do porte de Nathan Brown, fez este compatível com as versões 3.02 até a atual do Microchip MPLAB C18 compiler. Este manual irá auxiliá-lo a portar o µC/OS-II na sua versão mais atual, v2.86, utilizando o Microchip MPLAB C18 compiler na versão 3.02 até a mais atual, 3.30. Portanto deve-se ter este compilador instalado no computador, bem como o software MPLAB IDE.

A nova versão (v2.86) do Micrium µC/OS-II pode ser baixada no endereço http://www.micrium.com/products/rtos/ucos-ii_download.html, e é gratuita para fins acadêmicos. É preciso uma licença do software para utilizá-lo comercialmente.

1. Portando o µC/OS-II v2.86 para o Microchip PIC18F452

Para iniciarmos o porte do µC/OS-II v2.86 para o PIC18F452 devemos obter os arquivos fontes nos endereços:

A. http://www.sputnickonline.com/projects/programs/micro/uCOS_for_PIC18/files/MPLAB-C18v101.zip;

B. http://www.micrium.com/products/rtos/ucos-ii_download.html.

Com os arquivos fontes em mãos, siga os seguintes passos:

1. Extraia os arquivos MPLAB-C18v101.zip e Micrium-uCOS-II-V286.ZIP, obtidos dos endereços anteriores, em pastas separadas de sua preferência.

2. Na pasta extraída do arquivo Micrium-uCOS-II-V286.ZIP, crie uma nova pasta com o nome WORK (ou outro de sua preferência) no caminho *...\Micrium\SOFTWARE\uCOS-II*.

3. Da pasta extraída do arquivo MPLAB-C18v101.zip, copie os arquivos abaixo do caminho *...\SOFTWARE\uCOS-II\PIC18\MPLAB-C18\WORK* para o caminho *...\Micrium\SOFTWARE\uCOS-II\WORK* do passo 2: Estes serão os arquivos modificados por Nathan Brown que serão utilizados pelo nosso porte:

OS_CPU.H

OS_CPU_C.C

VECTORS.C

INCLUDES.H

uc-18f452.lkr

uc-18f452i.lkr

1ª OBS: A pasta extraída do arquivo MPLAB-C18v101.zip deve ser apagada para se evitar confusão nos caminhos de diretórios. A partir de agora só iremos modificar a

pasta extraída do arquivo Micrium-uCOS-II-V286.ZIP.

4. Renomeie os arquivos ***os_cfg_r.h*** e ***os_dbg_r.c*** do caminho *...\Micrium\SOFTWARE\uCOS-II\Source* para ***os_cfg.h*** e ***os_dbg.c*** respectivamente. Mova estes dois arquivos para o caminho *...\Micrium\SOFTWARE\uCOS-II\Work*.

2ª OBS: Modifique a propriedade de todos os arquivos fontes para que eles não estejam como somente leitura, permitindo assim que possam ser alterados.

5. Mova os arquivos ***ucos_ii.h***, ***ucos_ii.c***, ***os_core.c*** do caminho *...\Micrium\SOFTWARE\uCOS-II\Source* para o caminho *...\Micrium\SOFTWARE\uCOS-II\Work*.

6. No caminho *...\Micrium\SOFTWARE\uCOS-II\Work,* crie um arquivo (Documento de texto) de nome ***app_cfg.h***. Este arquivo servirá para guardar informações de configuração do seu projeto, como por exemplo prioridade das tarefas, tamanho das pilhas das tarefas e etc.

7. Adicione no arquivo ***OS_CPU.H,*** do caminho *...\Micrium\SOFTWARE\uCOS-II\Work,* os seguintes protótipos de função:

```
void OSStartHighRdy(void);
void OSIntCtxSw(void);
void OSCtxSw(void);
```

8. Substituir a diretiva #include <ucos_II.h> pela diretiva #include “INCLUDES.H” dos seguintes arquivos:

No caminho *...\Micrium\SOFTWARE\uCOS-II\Source:*

os_flag.c

os_mbox.c

os_mem.c

os_mutex.c

os_q.c

os_sem.c

os_task.c

os_time.c

os_tmr.c

No caminho *...\Micrium\SOFTWARE\uCOS-II\Work:*

os_core.c

ucos_ii.c

os_dbg.c

9. No arquivo ***ucos_ii.h*** (caminho *...\Micrium\SOFTWARE\uCOS-II\Work)*, na parte INCLUDE HEADER FILES do código, deletar/comentar as diretivas:

#include <app_cfg.h>

#include <os_cfg.h>

#include <os_cpu.h>

10. No caminho *...\Micrium\SOFTWARE\uCOS-II\Work*, no arquivo ***ucos_ii.c*** modificar o caminho dos arquivos nas diretivas **#include** da seguinte forma:

Deletar/comentar:

#include <os_core.c>

Substituir:

#include <os_flag.c>

#include <os_mbox.c>

#include <os_mem.c>

#include <os_mutex.c>

#include <os_q.c>

#include <os_sem.c>

#include <os_task.c>

#include <os_time.c>

#include <os_tmr.c>

Por:

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_flag.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_mbox.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_mutex.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_q.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_sem.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_task.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_time.c”

#include “…\Micrium\SOFTWARE\uCOS-II\Source\os_tmr.c”

11. Ainda no arquivo ***ucos_ii.c*** deletar/comentar diretiva **#define OS_GLOBALS**.

12. No arquivo INCLUDES.H (caminho *...\Micrium\SOFTWARE\uCOS-II\Work*) adicionar diretiva **#include "app_cfg.h"**.

13. No arquivo ***OS_CPU_C.C*** (caminho *...\Micrium\SOFTWARE\uCOS-II\Work*) substituir, ao longo de todo o arquivo, **TRUE** por **OS_TRUE.**

14. Ainda no arquivo ***OS_CPU_C.C*** substituir:

AARGB3

AARGB2

AARGB1

AARGB0

por:

__AARGB3

__AARGB2

__AARGB1

__AARGB0

ao longo de todo o arquivo.

15. No arquivo os_cfg.h (caminho ...\Micrium\SOFTWARE\uCOS-II\Work) configurá-lo com o mínimo de ferramentas do uCOS-II habilitadas. Os valores das constantes devem vir acompanhadas de L, conforme exemplo abaixo:

```
#define OS_TICKS_PER_SEC        100L
```

3ª OBS: Este é o arquivo de configuração do projeto do usuário, onde será determinado por ele quais as ferramentas necessárias para rodar sua aplicação. Este arquivo deve conter APENAS o necessário, já que ele é feito justamente para se economizar memória dos microcontroladores, visto que estes possuem pouca memória.

16. Devido ao problema de memória que os microcontroladores possuem, Nathan Brown sugere um artifício para economizar memória de dados (RAM). Para isso grava-se um pedaço do código fonte na memória ROM do microcontrolador. Esta alteração é apresentada a seguir:

No arquivo ***os_core.c***, na parte PRIORITY RESOLUTION TABLE do código, substituir a linha

INT8U const OSUnMapTbl[256] = {

por

/* Microchip PIC18xxx specific - lookup in program memory because of limited RAM space */

rom INT8U const OSUnMapTbl[256] = {

/* End Microchip PIC18xxx specific */

No arquivo ucos_ii.h, na parte GLOBAL VARIABLES do código, substituir a linha

extern INT8U   const   OSUnMapTbl[256];

por

/* Microchip PIC18xxx specific - lookup in program memory because of limited RAM space */

extern rom INT8U const   OSUnMapTbl[256];   /* Priority->Index   lookup table   */

/* End Microchip PIC18xxx specific */

17. Abra o software MPLA IDE da Microchip e crie um novo projeto indo em Project>New.

18. Dê um nome para seu projeto em Project Name e em Project Directory, escolha o caminho ...\Micrium\SOFTWARE\uCOS-II\Work. Confirme com OK.

19. Em Project>Select Language Tool Suite escolha como Active Toolsuite o item Microchip C18 Toolsuite. Localize cada um dos arquivos executáveis que aparecem em Toolsuite Contents na pasta em que foi instalado o Microchip MPLAB C18 compiler. Os arquivos estarão em um caminho parecido com *...\MCC18\bin*. Caso não tenha instalado o executável mpasmwin.exe, este pode ser desconsiderado. Confirme com OK.

20. Em Project>Set Languages Tool Location, vá em Microchip C18 Toolsuite>Default Search Paths & Directories>Include Search Path e coloque o diretório *…\MCC18\h* (caminho onde foi instalado o Microchip MPLAB C18 compiler). Em Microchip C18 Toolsuite>Default Search Paths & Directories>Library Search Path coloque o diretório *…\MCC18\lib*. Confirme com OK.

21. Em Project>Build Options>Project, vá na aba MPLAB C18. Configure da seguinte forma:

Na categoria *General*:

Integer promotion disabled;

Default storage class: Auto.

Na categoria Memory Model:

Large Code Model;

Large Data Model;

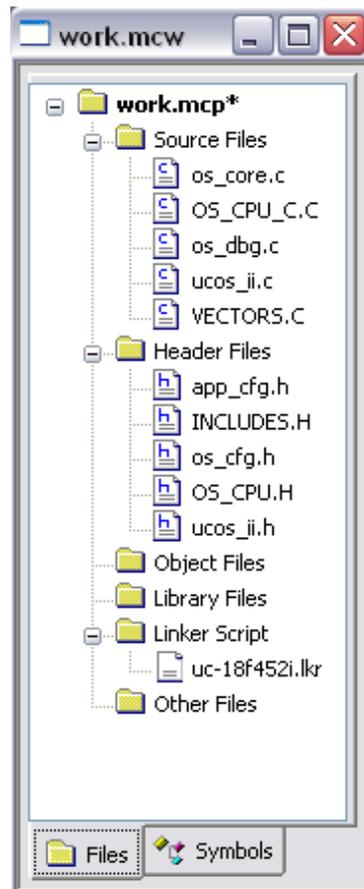
Multibank Model.

Na categoria Optimization:

Disable.

Confirme com Aplicar.

22. Ainda em Project>Build Options>Project vá à aba Directories. Clique em Suite defaults e confirme com OK.

23. Vá a View>Project. Adicione os arquivos fontes do µC/OS-II (localizados no caminho ...*\Micrium\SOFTWARE\uCOS-II\Work)*, clicando com o botão direito em cima de cada seção, da seguinte forma:

24. Coloque o arquivo do aplicativo do usuário no caminho *...\Micrium\SOFTWARE\uCOS-II\Work*) e adicione-o na seção **Source Files**. Configure o arquivo ***os_cfg.h*** com as ferramentas necessárias para rodar o aplicativo. Inclua os header files necessários para o aplicativo em **INCLUDES.H**. Compile o projeto em Project>Build all

Pronto. O kernel multitarefa de tempo real da Micrium µC/OS-II na sua última versão (v2.86) está pronto para rodar os aplicativos do usuário.

ANEXO - Aula 08. Desenvolvimento de Programas de Tempo Real: Metodologia.

Universidade Estadual de Campinas – UNICAMP
Faculdade de Engenharia Mecânica – FEM
Curso de Engenharia de Controle e Automação – MECATRÔNICA

Disciplina ES770 – Laboratório de Sistemas Digitais – 2o Semestre de 2008
Aula 08 – Desenvolvimento de Programas de Tempo Real: Metodologia
Autor: Luiz Otávio Saraiva Ferreira – 06/10/2008 – Arquivo: Roteiro_Aula08.odt

Introdução

Nesta aula será apresentada e exercitada a linguagem SDL (Specification and Description Language) de especificação e descrição de sistemas de controle a programa armazenado.

Teoria

A implementação de sistemas de controle por software, isto é, sistemas de controle a programa armazenado levaram ao desenvolvimento, nos anos 80, de várias metodologias específicas, sendo a SDL [1] uma das mais bem sucedidas. Baseia-se no conceito de máquinas de estado que se comunicam por sinais e têm processamento concorrente. É composta por uma variante gráfica (que estudaremos) e por uma variante textual. Há ferramentas comerciais que traduzem um diagrama SDL diretamente para um programa em C ou C++. A metodologia é muito simples, e será apresentada através de um exemplo completo que vai da especificação do problema ao código do programa em linguagem C. Os seguintes passos devem ser seguidos para a implementação de um sistema de controle a programa armazenado:

1. Redação de um texto descritivo do problema
2. Elaboração do Diagrama de Partição (com a estrutura funcional do sistema)
3. Diagrama de Interação entre Blocos (com a comunicação entre os blocos)
4. Diagramas SDL (que descrevem as máquinas de estados e suas interações)
5. Programação e depuração.

Nosso exemplo será um relógio contador muito simples, cuja especificação é dada a seguir.

Passo #1: Texto Descritivo do Problema

Projetar e programar um contador crescente/decrescente de 4 dígitos a ser implementado no kit MODULO 2. Os dois dígitos deverão ser exibidos no display de

7 segmentos. A contagem deverá ter incremento ou decremento a 4 Hz. Acionamento via Teclado:

- Contagem crescente - chave S1 pressionada; sinal b1;
- Contagem decrescente - chave S2 pressionada; sinal b2;
- Pára contagem - chave S3 pressionada; sinal b3.

Para deixar mais claro o comportamento desejado, um diagrama de estados é mostrado abaixo:

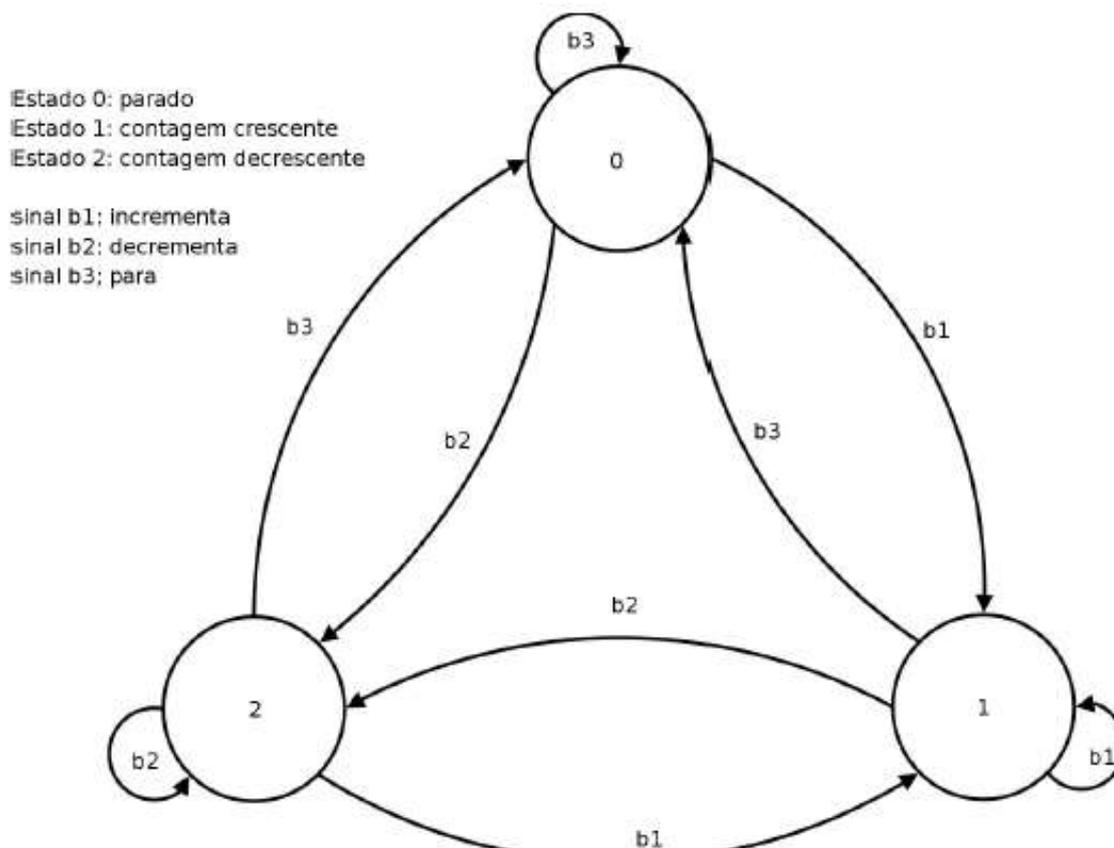



Figura 1: Diagrama de estados do contador crescente/decrescente. As transições de estado são disparadas pelo pressionar das chaves S1, S2 e S3 do kit MODULO 2.

Passo #2: Elaboração do Diagrama de Partição

As funções do sistema são identificadas e divididas em partes mais simples, até que se chegue a um nível de complexidade suficientemente pequeno para solução de cada parte do problema. Geralmente os sistemas podem ser divididos nos seguintes grupos de funções: Entradas, Saídas,Temporizações e Aplicação, que no nosso caso é a contagem. Os blocos funcionais resultantes do processo de análise são as caixas inferiores do diagrama de partição. Serão doravante chamadas de PROCESSOS do sistema, e cada processo será uma máquina de estados.

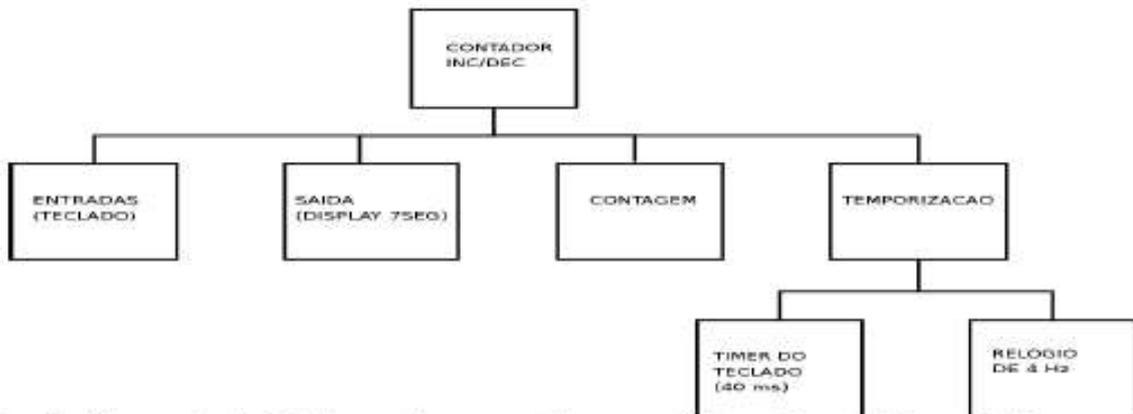



Figura 2: Diagrama de Partição do contador crescente/decrescente. Notar o formato de árvore do diagrama, em que os blocos do nível de cima são detalhados no nível de baixo. A temporização contém dois PROCESSOS: o timer do teclado e o relógio de 4 Hz.

Passo #3: Elaboração do Diagrama de Interação entre Blocos

Dados os processos do sistema obtidos no passo anterior, imagina-se agora como esses processos interagirão entre si, isto é, que sinais um mandará para o outro informando o da ocorrência de eventos que provocarão transições de estados. Importante dizer-se que, por convenção, os sinais são binários, isto é, de apenas um bit, e são imediatamente consumidos ao serem reconhecidos pelos processos que os recebem. Mas pode haver dados associados a um sinal. O diagrama abaixo mostra as interações entre os blocos do contador.

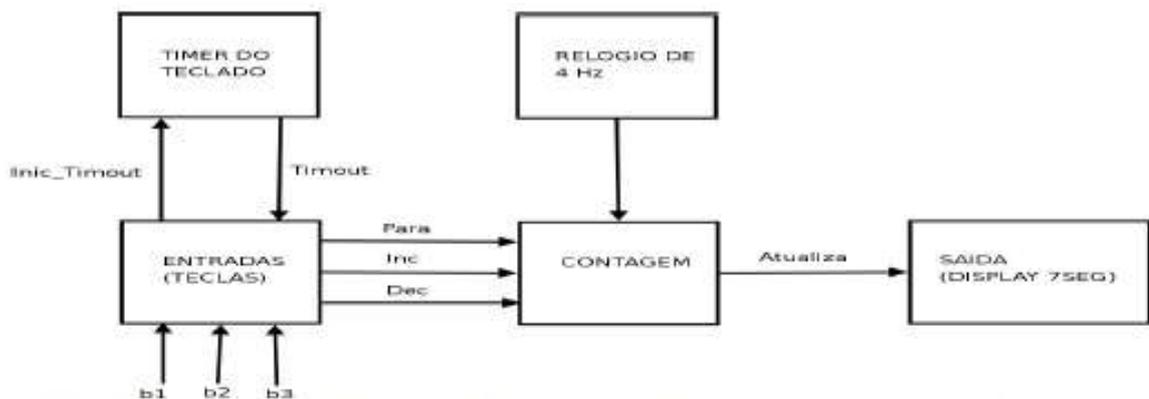



Figura 3: Diagrama de interação entre blocos funcionais do contador crescente/decrescente. O teclado gera os sinais b1, b2 e b3, que são enviados para o bloco TECLAS. Ao receber um sinal de teclado, o processo TECLAS envia ao processo TIMER DO TECLADO um sinal ordenando o início da temporização de eliminação do repique do teclado. Quando transcorreu o tempo de repique do teclado, seu TIMER envia o sinal TIMOUT para o processo TECLAS. Quando o processo TECLAS recebe o TIMOUT, checa o estado das teclas e, se estas estiverem estáveis, gera os sinais correspondentes (Pára, Inc, Dec) para o processo CONTAGEM. O processo contagem faz um incremento ou decremento do contador a cada chegada do sinal de TIC do relógio de 4 Hz, quando então envia um sinal (Atualiza) para o processo DISPLAY 7SEG, responsável pela atualização valor de contagem exibido no display de 7 segmentos.

Passo #4: Diagramas SDL

Cada processo é então transformado numa máquina de estados, usando-se para isso o formato gráfico da SDL, cujos símbolos básicos são mostrados abaixo.

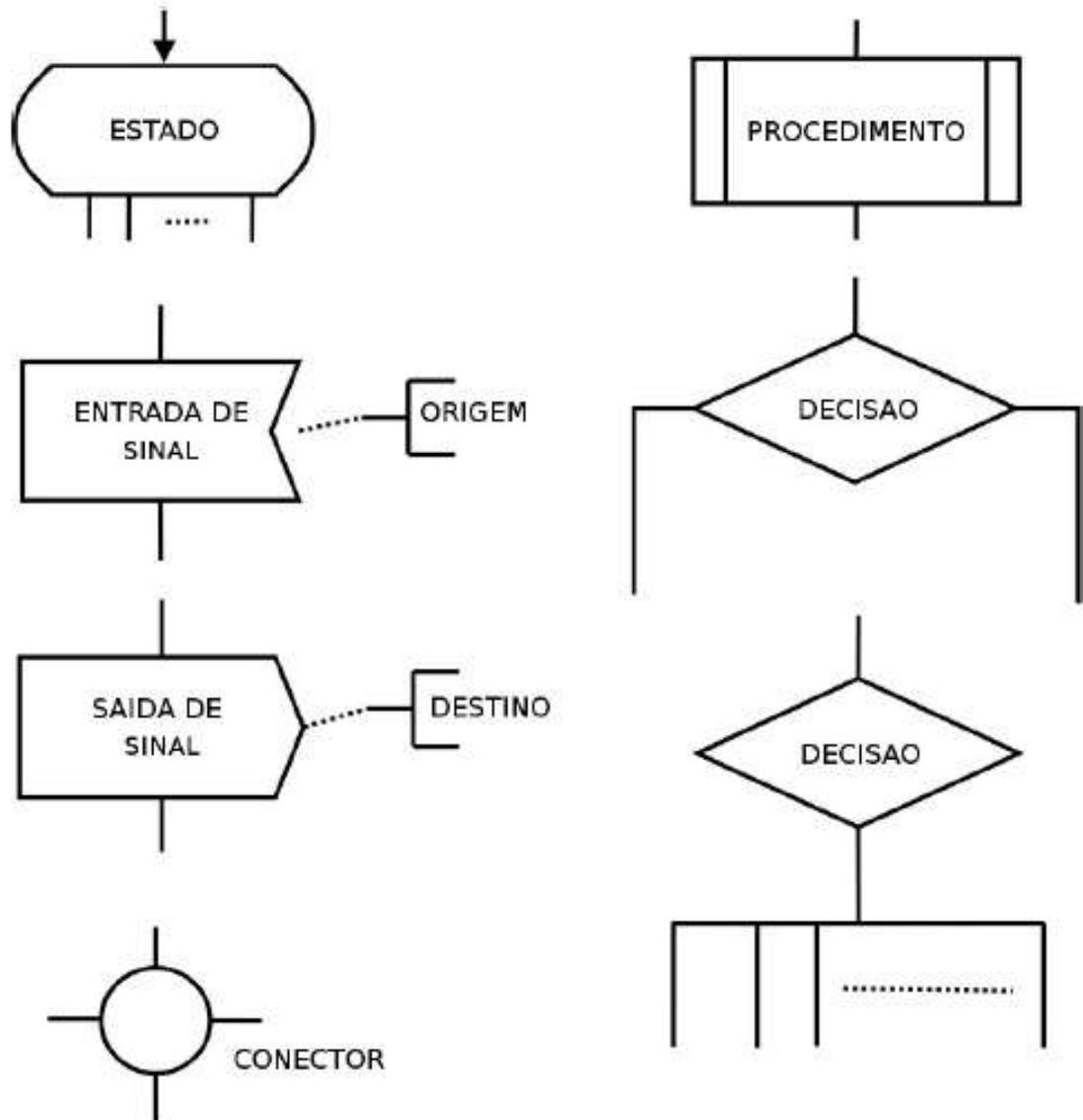



Figura 4: Símbolos básicos da SDL gráfica, usados para representação das máquinas de estados correspondentes aos PROCESSOS do sistema em especificação.

Serão mostrados a seguir os diagramas SDL das diversas máquinas do nosso contador.

Diagrama SDL do processo CONTADOR.

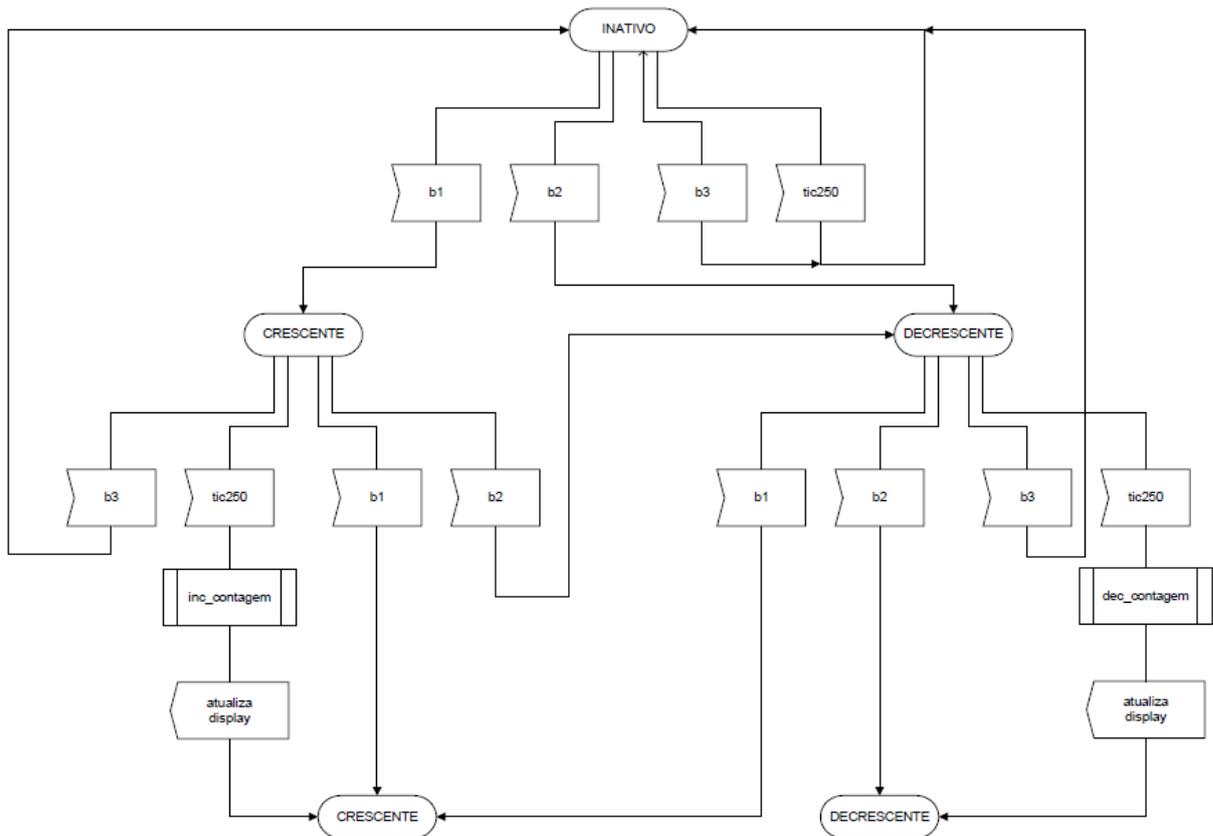



Figura 5: Diagramas SDL do processo CONTADOR. Notar sua correspondência com a máquina de estados mostrada anteriormente, no Passo #1.

Diagrama SDL dos processos TECLADO e TIMER

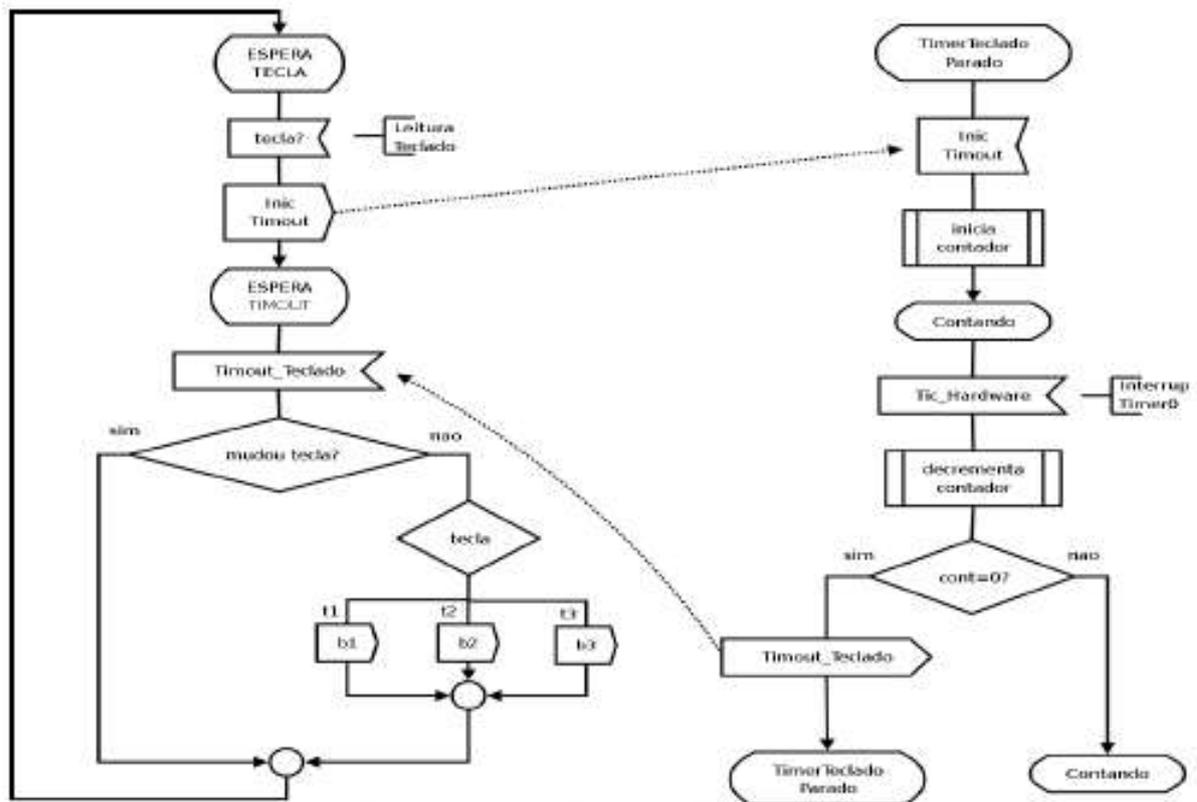



Figura 6: Diagramas SDL dos processos TECLADO e TIMER. Notar as linhas tracejadas indicando o trajeto dos sinais.

Diagrama SDL dos processos DISPLAY e RELÓGIO.

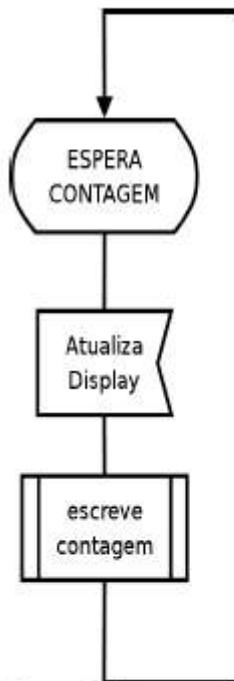



Figura 8: Diagrama SDL do processo DISPLAY.

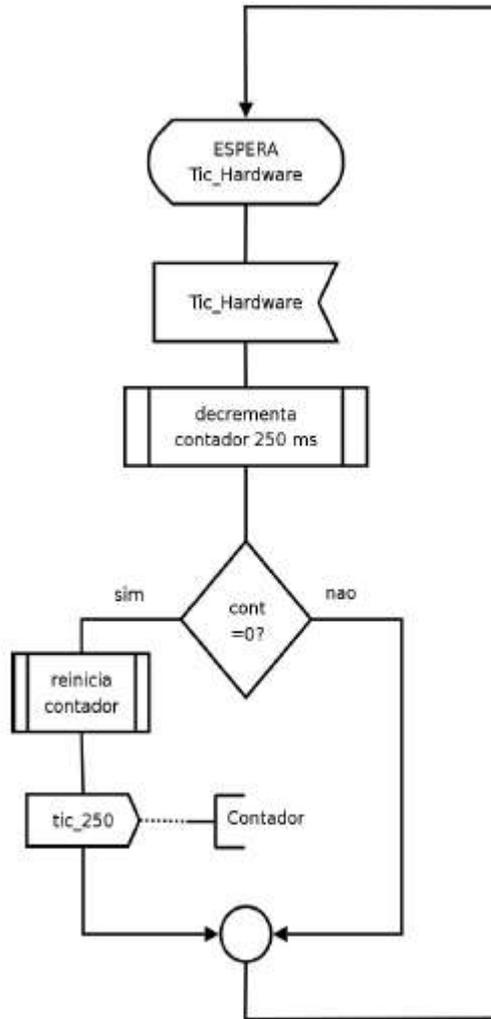



Figura 7: Diagrama SDL do relógio de 4 Hz.

Antes de passarmos aos fluxogramas do código que implementará nossas máquinas de estado, é necessário apresentarmos uma peça de software fundamental para o sistema: o ESCALONADOR. Esse software é encarregado de gerenciar a distribuição do tempo do processador aos processos. Geralmente é o coração dos sistemas operacionais. Como não temos um sistema operacional, implementaremos nosso próprio escalonador, que será do tipo mais simples possível: o Escalonador Cíclico. Esse tipo de escalonador se caracteriza por passar o controle do processador a uma lista de processos, sempre na mesma ordem de chamada, a um intervalo de tempo fixo. No nosso caso o Escalonador será uma função ativada pela interrupção do TIMER0. A cada interrupção do TIMER0 o Escalonador chamará todas as funções que implementam as máquinas de estado do nosso sistema, conforme pode ser visto na Figura 9.

Fluxograma do Escalonador Cíclico

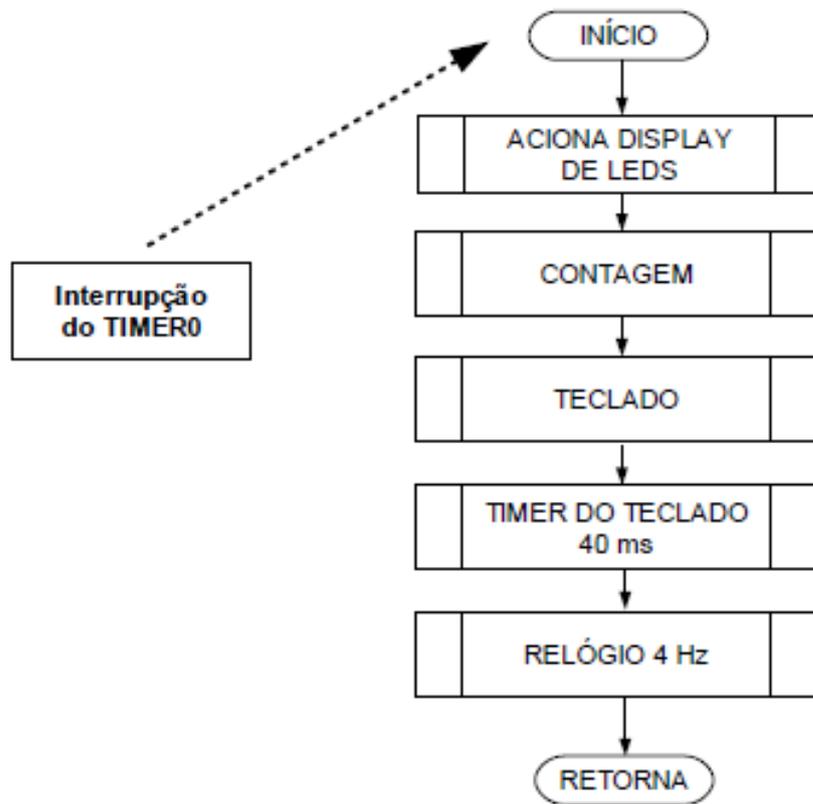



Figura 9 – Fluxograma do Escalonador Cíclico

Passo #5: FLUXOGRAMAS DOS BLOCOS

Fluxograma do processo Aciona Display Leds

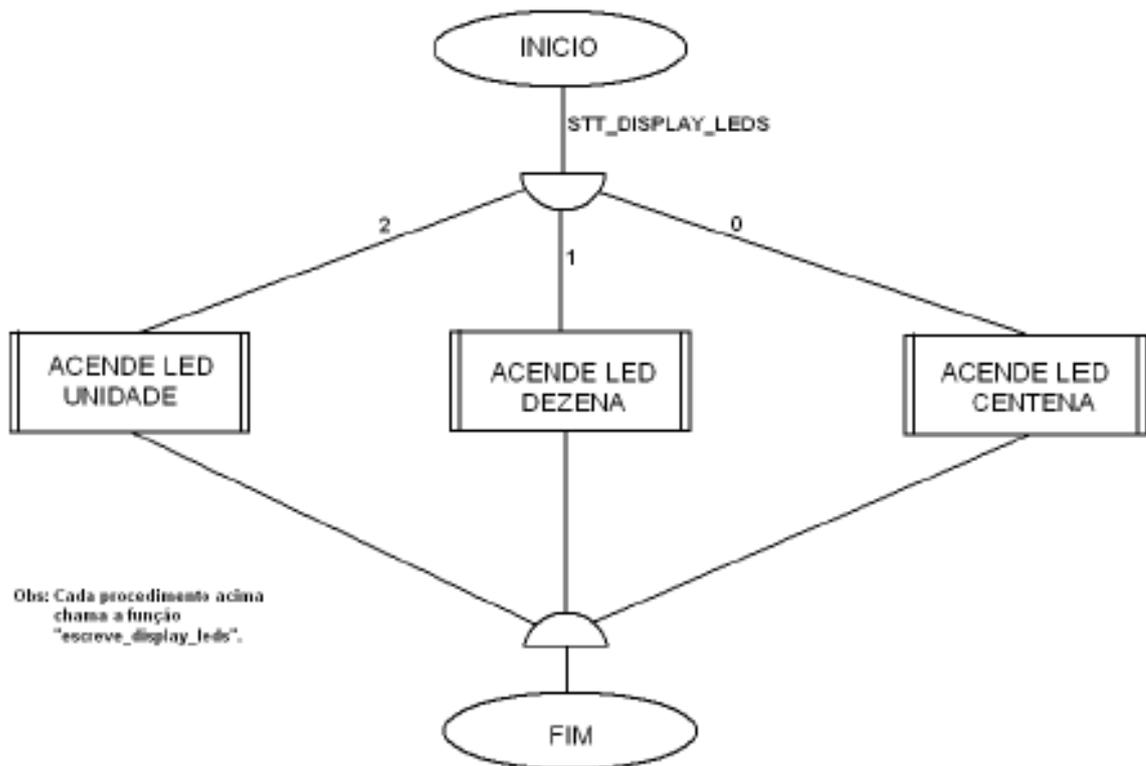



Figura 10 – Fluxograma Aciona Display Leds

Fluxograma do processo Contagem

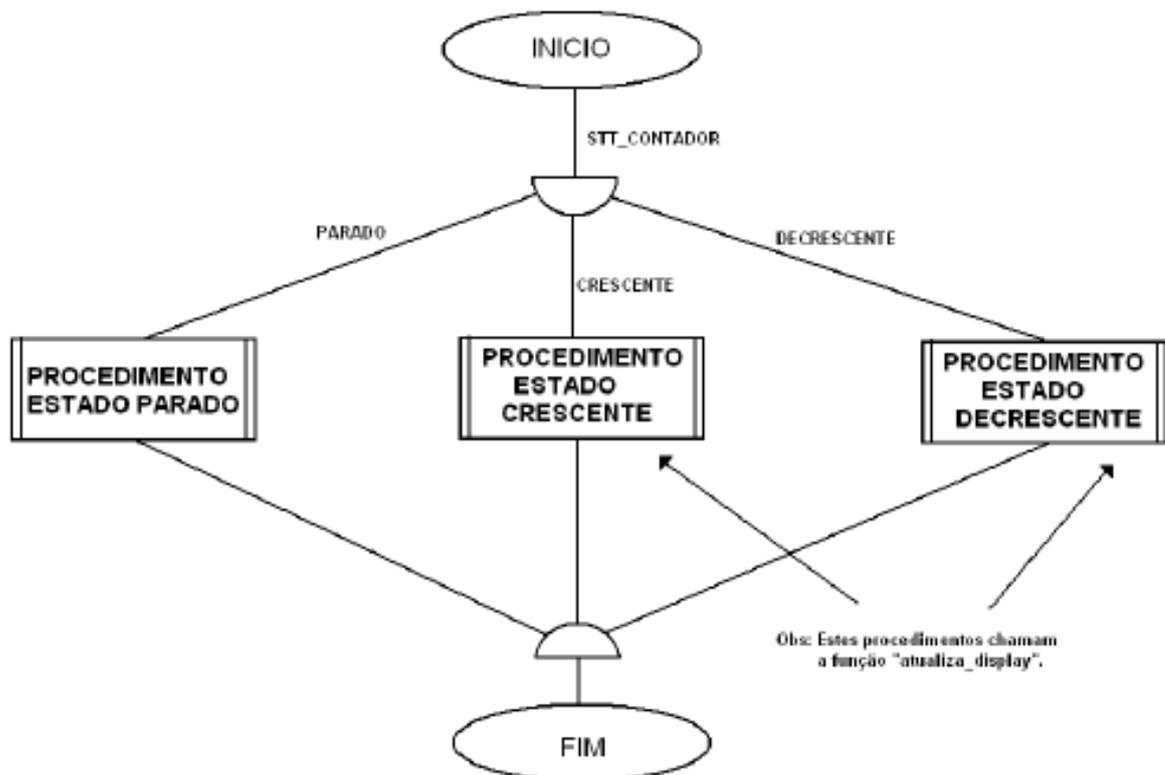



Figura 11 – Fluxograma Contagem

Fluxograma do processo Teclado

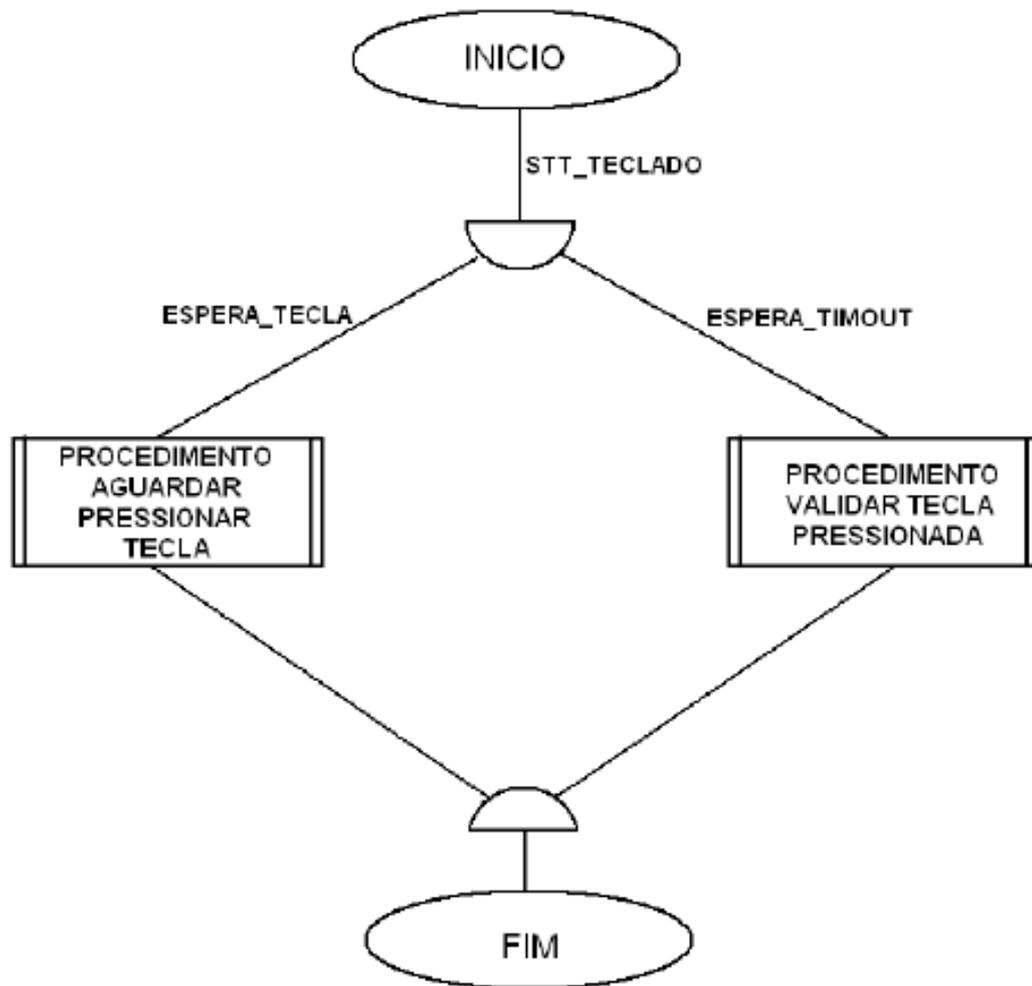
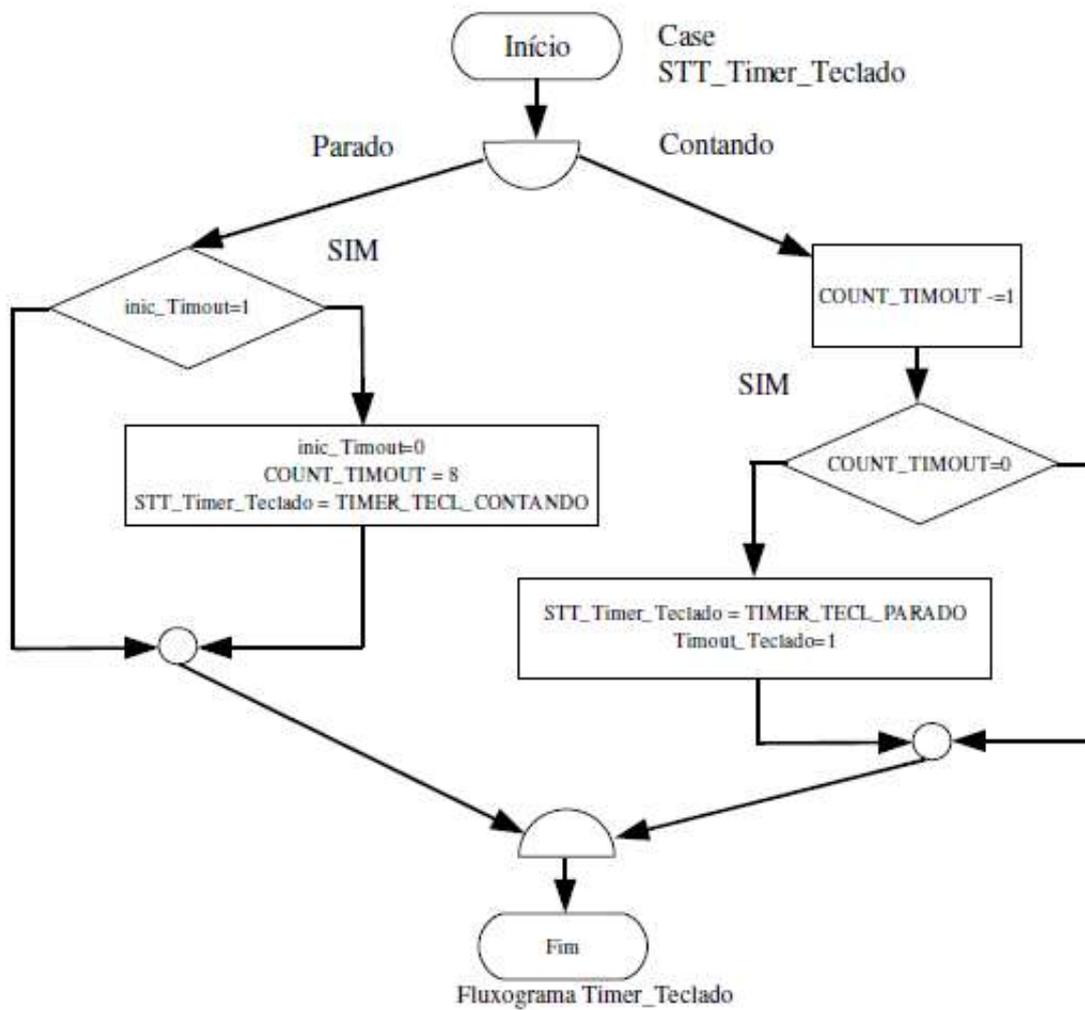



Figura 12 – Fluxograma Teclado

Fluxograma do processo Timer_Teclado

Figura 13 – Fluxograma Timer_Teclado

Fluxograma do processo Relógio 4 Hz

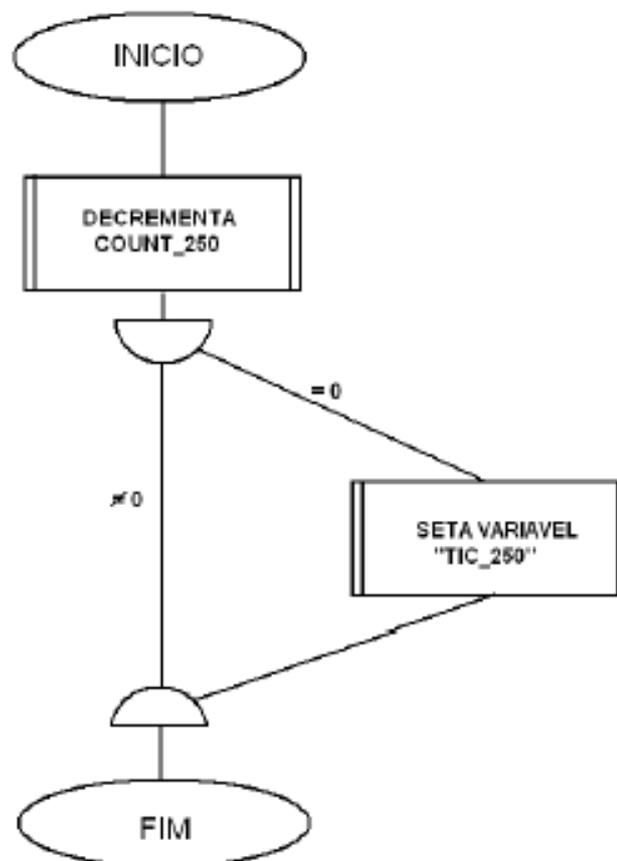



Figura 14 – Fluxograma Relógio 4 Hz

Exercício em sala

Acrescente a função ZERAR CONTADOR, acionada pelo botão S4 do kit MODULO 2.

Exercício extraclasse

Passe o programa do controlador proporcional de velocidade do ventilador, dado na aula passada, para a forma de máquinas de estados. Utilize as chaves S1 e S2 para incrementar e decrementar o valor do ganho proporcional KP. Mostre o valor de KP no display de sete segmentos. Prazo de entrega: duas semanas.

[1] Belina, Ferenc, SDL with applications from protocol specification, Prentice Hall International (UK); Municha: Hanser Verlag, 1991.
[2] Manual da linguagem SDLRT, extensão de tempo real da SDL. Pode ser baixado gratuitamente do link http://www.sdlrt.org/standard/V2.2/pdf/SDLRT.pdf; 13/09/2006, 15h00min.